Numerical Methods in Physics (515.421)

Numerische Methoden in der Physik (515.421)

Prof. Heinrich Sormann
Institut für Theoretische Physik - Computational Physics
TU Graz

English translation by: Dr. Lilia Boeri, ITP-CP, TU Graz

Script WS 2013/2014

Computers are useless.
They can only give you answers.

Attributed to Pablo Picasso...

Note:

The script for the lecture "Numerical Methods in Physics", WS 2013/2014, has been translated into english from the original script of Prof. Heinrich Sormann by Dr. Lilia Boeri. The present english version contains only chapters covered in the course of the main lecture in WS 2013/14.

The list below shows the correspondence between the chapters of the German (D) and English (E) script.

D(1): Einführung.
E(1): Introduction.

D(2): Numerische Methoden zur Lösung linearer, inhomogener Gleichungssysteme.
E(2): Numerical Methods for Linear, Inhomogeneous Systems of Equations.

D(4): Least-Squares Approximation.
E(3): Least Squares Approximation.

D(5): Numerische Lösung von trascendenten Gleichungen.
E(4): Numerical Solution of Transcendental Equations.

D(7): Eigenwerte und Eigenvektoren reeller Matrizen.
E(5): Eigenvalues and Eigenvectors of Real Matrices.

D(8): Numerische Methoden zur Lösung von gewöhnlichen Differentialgleichungen: Anfangswertprobleme.
E(6): Numerical Methods for Ordinary Differential Equations: Initial value problems.

Chapter 1

Introduction

1.1 Basics.

The subject of this lecture on "*Numerical Methods in Physics*" is the solution of real physical problems with methods of numerical mathematics. These methods permit to treat problems which are so complex that they can only be solved with the aid of a computer.

A good definition of *numerical mathematics* can be found, for example, in [1], page 8:

Numerical mathematics concerns the solution of mathematical problems using numerical calculations. This term indicates a finite sequence of basic mathematical operations, such as addition, subtraction, multiplication and divisions, using finite-digits numbers.

[1] contains also a nice schematical illustration of the basic methodology of numerical mathematics and of all the possible unavoidable errors that can affect the results.

```
|--------------------|       |------------------------------|
| Input Information  | ===>  |        Input error           |
|       (Input)      |       |  (Measurement or roundoff    |
|                    |       |   error)                     |
|--------------------|       |------------------------------|
          |                                 |
          |                                 |
|--------------------|       |----------------------------------------|
|    Algorithm       | ===>  |             Algorithmic error          |
|                    |       | (Methodological error; roundoff error) |
|--------------------|       |----------------------------------------|
          |                                 |
          |                                 |
|--------------------|       |------------------------------|
| Output Information | ===>  |         Output error         |
|      (Output)      |       |       (Roundoff error)       |
|--------------------|       |------------------------------|
```

A schematic diagram illustrating the methodology of numerical mathematics, from [1].

A central concept in numerical mathematics is that of *algorithm.* This term, according to [2], page 9, indicates:

A finite number of precise instructions, which require specific input data and, executed in a given sequence, determine the final solution.

1.2 Errors: general considerations

As shown from the previous diagram, all stages of a numerical calculations (data input; actual calculation process; data output) can lead to errors in the results. Typical causes are the eventual *roundoff* and *methodological* errors, which we will explain in detail in this chapter.

The results of computer calculations are (with a very few exceptions) subject to errors, *i.e.* they do not coincide exactly with the "true" solution of the problem.

Therefore, it is of outmost importance, when working with a computer, to keep under control all possible errors and to be able to estimate the precision of the results that are obtained.

On the subject "errors in numerical methods", we refer the reader to the relevant chapters from [7] and [8].

1.2.1 Absolute and relative errors. Machine precision.

Before discussing in more detail the various types of errors, we think that it is useful to introduce the concept of *absolute* and *relative* error:

If x is the (unknown) real value of a quantity and $\bar{x}$ is the relative approximate value obtained with a numerical calculation, we define:

$$\epsilon_a = x - \bar{x} \tag{1.1}$$

absolute error and

$$\epsilon_r = \frac{\epsilon_a}{\bar{x}} = \frac{x - \bar{x}}{\bar{x}} \tag{1.2}$$

relative error of $\bar{x}$.

In practical cases, it is usually preferrable to employ the relative error, given by formula (1.2). The reason is clearly illustrated by the following simple example:

	x	$\bar{x}$	ϵ_a	ϵ_r
(a)	0.1	0.09	0.01	≈ 0.1
(b)	1000.0	999.99	0.01	≈ 0.00001

Structure chart 1 — Determination of the parameter τ

```
tau:=1.0
wold:=1.0
    tau:=tau/2.0
    wnew:=wold + tau
wnew=wold
print: 2*tau
```

It is immediately clear that the approximation in (b) is much more satisfactory than in (a), although the absolute error is the same in both cases.

Another fundamental concept is that of **machine precision**: this is the smallest (positive) number τ, for which:

$$1 + \tau > 1$$

For a (non-existing) supercomputer that can save real numbers with arbitrary accuracy, the above inequality would be satisfied for all values of τ. In real computers, however, τ has a finite value, which it is very important to know. It can be estimated in practice using the algorithm shown in the structure chart 1.
The following results were obtained on a PC IBM (the same is true for all tests contained in this script, unless otherwise specified):

	Bytes	τ	Number sign. digits
C-float	4	$1.19 \cdot 10^{-7}$	7
C-double	8	$2.22 \cdot 10^{-16}$	16
F90-real	4	$1.19 \cdot 10^{-7}$	7
F90-double prec.	8	$2.22 \cdot 10^{-16}$	16
Pascal-single	4	$1.19 \cdot 10^{-7}$	7
Pascal-real	6	$1.82 \cdot 10^{-12}$	12
Pascal-double	8	$2.22 \cdot 10^{-16}$	16
Pascal-extended	10	$1.08 \cdot 10^{-19}$	19

Realization of the structure chart 1 in C and F90:

```
// C-PROGRAM FOR THE CALCULATION OF MACHINE PRECISION

#include <iostream.h>

void main()
{
  float tau,wold,wnew;

  tau=1.0;
  walt=1.0;
  do {
    tau/=2.0;
    wnew=wold+tau;
  } while (wnew != wold);
  cout <<"TAU = "<<2*tau<<"\n";
}
```

```
PROGRAM machine

! F90-Program for the calculation of machine precision

IMPLICIT NONE

REAL tau,wold,wnew

tau=1.0
wold=1.0

DO
  tau=tau/2.0
  wnew=wold+tau
  IF(wnew .EQ. wold)EXIT
END DO

PRINT '(" tau = ",E12.6)',2.0*tau

END PROGRAM machine
```

1.2.2 Input errors. Ill-conditioned problems.

Input errors (i.e. *inherent errors*) are uncertainities in the input data with which the computer performs the calculation. These uncertainities can have several origins, i.e. experimental measuring errors, or *roundoff* errors.

If the results of a calculation are strongly influenced by this input errors we speak of an "ill-conditioned problem".

As an example of an ill-conditioned linear set of equations, we consider the example in [7], page 97:

$$x + 5.0y = 17.0$$

$$1.5x + 7.501y = a$$

We assume that a is determined by an experiment, and thus affected by uncertainity:

$$a = 25.503 \pm 0.001$$

It is easy to verify that the exact solutions for this system for $a = 25.503$ are: $x = 2.0$ and $y = 3.0$.
The table below shows how the values of x and y change, if we assume that the last digit of a changes by one unit:

a	x	y
25.503	2.	3.
25.502	7.	2.
25.504	-3.	4.

Obviously, this set of equations constitutes an extremely ill-conditioned problem! The results, given the experimental uncertainity on a, are completely meaningless!

1.2.3 Algorithmic errors.

are errors, which occur during the evaluation of the calculation specification. The causes are either *roundoff* errors or *methodological* errors.

1.2.4 Methodological errors.

Methodological errors occur when the original mathematical problem is replaced by a simplified problem.
An example is the numerical integration:

Let us consider the definite integral

$$I = \int_{x=1}^{2} \frac{dx}{x^2}$$

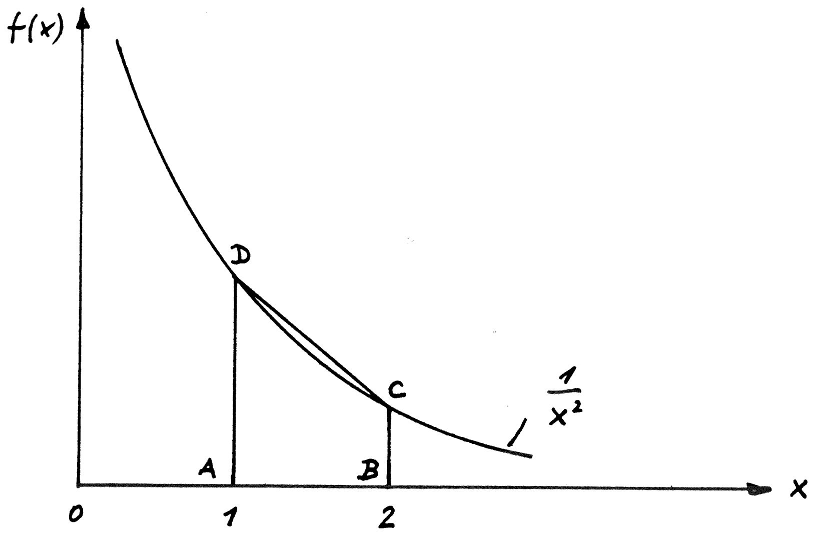


Figure 1.1: Principle of numerical integration

which admits an exact solution $I = 0.5$. Let us assume that we evaluate the same integral numerically, approximating the "true" integral surface with the trapezoidal surface $ABCD$ (Fig. 1.1):
The 'numerical' result is in this case 0.625, i.e. we have introduced an (absolute) methodological error:

$$\epsilon_p = 0.5 - 0.625 = -0.125.$$

Other typical methodological errors arise for example from the truncation of an infinite series (*truncation error*), and so on.

1.2.5 Roundoff errors.

Roundoff errors are unavoidable[1], and derive from the fact that in a computer only a limited number of bits is available to represent a real number (or, better, its *Mantissa*).[2]

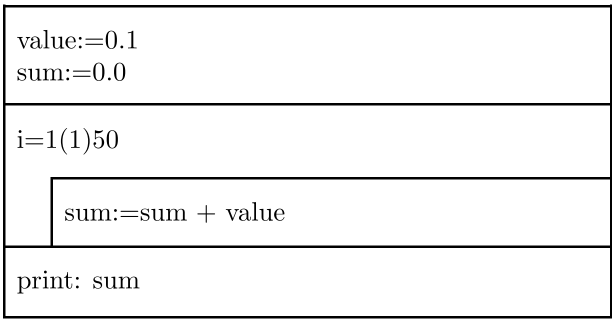
A simple example:

```
 Calculation "by Hand":            Computer (7 sign. digits):

   123456.789                        0.1234568E+06
+        9.876543                 +  0.9876543E+01
----------------                  ----------------
   123466.665543                     0.1234667E+06
```

[1] 'unavoidable' means that, as we will demonstrate in the following, there is no possibility to reduce roundoff errors!

[2] This excludes the very rare case in which one deals with an INTEGER arithmetic.

Structure chart 2 — Demonstration of the effect of roundoff errors



```
value:=0.1
sum:=0.0
i=1(1)50
    sum:=sum + value
print: sum
```

Another example: If we perform the calculation sketched in the structure chart 2 on a PC, we encounter a phenomenon which is hard to understand at first sight:
The result is we get is 4.999998, i.e. we see the effect of a roundoff error, athough each of the 50 constant factors 0.1 which are summed up is saved correctly in memory.

The reason is that the number 0.1, which is representable without problems in a decimal system, in the binary system used by computers is a periodic number:

$$0.1_{10} = 0.000110011001100\ldots_2$$

1.3 Methodological and roundoff errors.

In the following, we will see typical examples of different types of errors which occur in computer programs. We will also discuss possible ways to recognize and minimize the effects of methodological and roundoff errors.

1.3.1 Connection between roundoff errors and algorithm.

The first example shows that the roundoff error on the evaluation of a mathematical expression can be reduced through an appropriate reformulation of the algorithm.

Let us imagine we wish to evaluate numerically the following expression:

$$F(x) = \frac{\sqrt{1+x} - \sqrt{1-x}}{x}$$

for $x << 1$. Since $F(x)$ has a well-defined analytical expression, in this case there is no methodological error. Furthermore, we can assume that in this case there are no input or output errors.

If we code this formula without further reshaping, i.e.

$$F := (SQRT(1. + X) - SQRT(1. - X))/X \quad ,$$

we obtain the following result for decreasing values of x:

Tab.1.1: Numerical evaluation of F(x) and F1(x).

x	F	F1
10^0	$0.1414214E+01$	$0.1414214E+01$
10^{-1}	$0.1001256E+01$	$0.1001256E+01$
10^{-2}	$0.1000013E+01$	$0.1000013E+01$
10^{-3}	$0.1000041E+01$	$0.1000000E+01$
10^{-4}	$0.1000153E+01$	$0.1000000E+01$
10^{-5}	$0.1001357E+01$	$0.1000000E+01$
10^{-6}	$0.1013279E+01$	$0.1000000E+01$
10^{-7}	$0.1192093E+01$	$0.1000000E+01$
10^{-8}	$0.0000000E+01$	$0.1000000E+01$

Two features immediately catch our attention in the values of F shown in the above table:

- The value of the function does not tend to 1 for $x \to 0$, as it should, but begins to increase at $x = 10^{-2}$.
- For values of x smaller than $x = 10^{-8}$, F is exactly zero.

This behaviour can be easily explained:

- For small x the two square roots $\sqrt{1+x}$ and $\sqrt{1-x}$ represent two *almost equal numbers*. Subracting these numbers from each other, and dividing by x, the two roundoff errors, that are by themselves small, give a large effect, due to the law of error propagation ('subtractive cancellation').
- For $x < \tau$, we have that both $1 + x = 1$ and $1 - x = 1$; therefore, the value of $F(x)$ is exactly zero.

So far, we have only identified the error. The cure in this case is very simple: If we rewrite $F(x)$ as:

$$F1(x) \equiv F(x) = \frac{(\sqrt{1+x} - \sqrt{1-x})}{x} \cdot \frac{(\sqrt{1+x} + \sqrt{1-x})}{(\sqrt{1+x} + \sqrt{1-x})} = \frac{2}{\sqrt{1+x} + \sqrt{1-x}}$$

we recover the correct results (see Table 1.1).

Another example ([9], page 178) shows that we can end up in trouble also in case of seemingly very simple problems, if we program the formulas without thinking.

Let us consider the numerical solution of a second-order equation:

$$ax^2 + bx + c = 0$$

with real coefficients a, b, and c.

The well-known solutions are

$$(a) \quad x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \qquad (b) \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

We immediately see that for very small values of the coefficients a and/or c there is the danger of a 'subtractive cancellation' on the values of x_1 and x_2 (for $b > 0$ and $b < 0$ respectively).

But it is also easy to recast the above expression in the equivalent form:

$$(c) \quad x_1 = \frac{2c}{-b - \sqrt{b^2 - 4ac}} \qquad (d) \quad x_2 = \frac{2c}{-b + \sqrt{b^2 - 4ac}} \quad .$$

From the above considerations, it is clear that expressions (c), (b) and (a), (d) are stable against roundoff errors for $b > 0$ and $b < 0$ respectively.

The following algorithm sums up the above expressions:

$$x_1 = \frac{q}{a} \qquad x_2 = \frac{c}{q}$$

with

$$q \equiv -\frac{1}{2}\left[b + sgn(b)\sqrt{b^2 - 4ac}\right] \quad .$$

1.3.2 Roundoff and methodological errors in numerical differentiation.

The basic principle of a numerical differentiation (derivation) is that of replacing the required differential with a suitable finite difference expression, i.e.

$$\frac{d}{dx} f(x) \mid_{x=x_o} \approx \frac{f(x_o + h) - f(x_o)}{h}. \tag{1.3}$$

The differential ratio (1.3) converges to the first derivative of the function $f(x)$ in x_o when the increment $h \to 0$. A *finite* increment (or *stepsize*) h introduces a *methodological error* ϵ_V, which gets smaller and smaller with decreasing h.
Theoretical considerations show that ϵ_V must have the form

$$\epsilon_V = C_V(h) \cdot h \quad , \tag{1.4}$$

where the quantity $C_V(h)$ in many cases depends only weakly on h, and can be replaced in practice by a constant:

$$C_V(h) \approx C_V \quad .$$

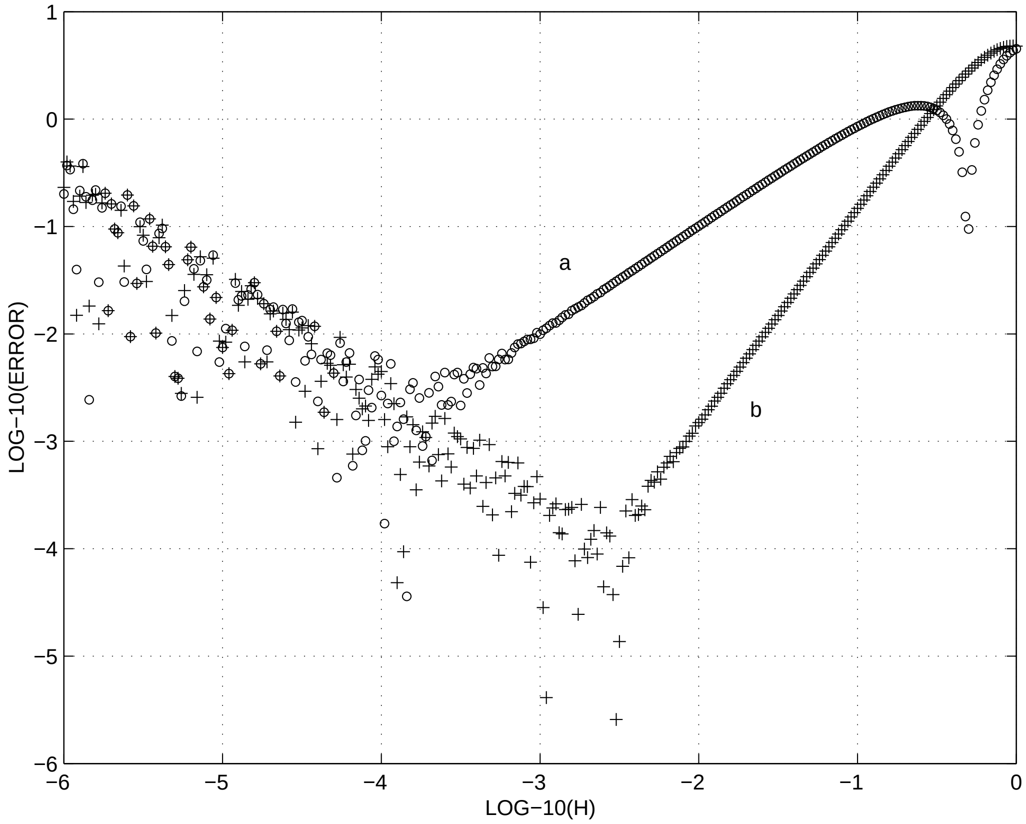


Figure 1.2: Error diagram for numerical differentiation (a), formula (1.3), (b) Formula (1.7)

If we plot ϵ_V vs h in a *log-log* scale, the *error curve* satisfies the linear relation

$$\log \epsilon_V = \log C_V + \log h \tag{1.5}$$

with slope +1.

The curve (a) in Fig. 1.2 shows these relations for the concrete example

$$f(x) = \ln x \cdot \sin(5x)$$

$$\frac{d}{dx} f(x) \mid_{x_o=3} = \frac{\sin(15)}{3} + 5 \ln 3 \cos(15) \quad .$$

Clearly the relation (1.4) is well fulfilled in the interval $10^{-3} < h < 10^{-1}$. For increments that are larger or smaller than these values, the 'real' behaviour of the error deviates largely from this behaviour.

The causes for this behaviour are the following:

- The deviation in the interval $h > 10^{-1}$ is due to the fact that, if the increment is too large, C_V is not constant.
- It is more interesting to analyze the behaviour of the error in the interval $h < 10^{-3}$, where the error does not follow the expected linear

behaviour with h – Eq. (1.4). On the other contrary, it acquires a strong functional dependence, and this is due to a strong "subtractive cancellation" of the roundoff errors in (1.3).

The dependence of the roundoff error on h does not follow a smooth curve, but tends to the expression:

$$\epsilon_R = \frac{C_R}{h} \quad . \tag{1.6}$$

From what we said so far, we can conclude:

- Using an *optimal* increment step h_{opt} we can reduce the error down to a minimal value which *cannot be further reduced* – in concrete examples this error is around $5 \cdot 10^{-3}$.
- Using an increment step $h < h_{opt}$ does not improve the numerical result, but - on the contrary - makes it worse!

There are two possible strategies to further reduce this minimal error:

- The roundoff error can be reduced increasing the precision with which real numbers are stored in memory (single → double; float → double etc.).
- The methodological error can be reduced using an *improved algorithm*, for example using the more efficient formula

$$\frac{d}{dx} f(x) \mid_{x=x_o} \approx \frac{f(x_o + h) - f(x_o - h)}{2h} \tag{1.7}$$

  for the differential ratio. Since in this formula ϵ_V decreases much faster with decreasing h as compared to expression (1.3), i.e.

$$\epsilon_V = \bar{C}_V(h) \cdot h^2 \quad , \tag{1.8}$$

  but the roundoff error has the same dependence on h as before, the minimal error can be reduced by roughly one order of magntidude – see curve (b) in Fig. 1.2.

In summary, we can say:

The error diagnostics in this example (and in many areas of numerical mathematics) is largely simplified by the fact that under certain assumptions (in this case, in a particular range of h) the methodological error is much larger than the roundoff error.

1.3.3 Error diagnostics in the numerical evaluation of the error function.

In many applications one has to evaluate numerically the function

$$\mathrm{erfc}(x) = 1 - \mathrm{erf}(x),$$

where $\mathrm{erf}(x)$ represents the *Gaussian error function.*

The integral representation, i.e. the representation in form of a Taylor series is:

$$\mathrm{erfc}(x) = 1 - \frac{2}{\sqrt{\pi}} \int_{z=0}^{x} dz e^{-z^2} = 1 - \frac{2}{\sqrt{\pi}} \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{n!(2n+1)} \quad .$$

The numerical evaluation of $\mathrm{erfc}(x)$ can be done using this Taylor series, whose convergence in principle is assured for all real arguments x:

$$\mathrm{erfc}(x) \approx 1 - \frac{2}{\sqrt{\pi}} \sum_{n=0}^{nmax} a_n \quad \textit{with} \quad a_n = \frac{(-1)^n x^{2n+1}}{n!(2n+1)} \quad .$$

INSET: Numerical evaluation of a Taylor series

- From the point of view of computational efficiency it is extremely inconvenient to evaluate each term a_n of the series independently. It is much better to calculate each a_n from the previous a_{n-1}. In fact, to calculate the ratio a_n/a_{n-1}, we can exploit the *recursive* relation

  $$a_n = \left[-\frac{x^2(2n-1)}{n(2n+1)} \right] a_{n-1} \qquad \text{for} \quad n = 1, 2, \ldots$$

  with

  $$a_0 = x\,.$$

- Truncating the series at $n = n_{max}$ introduces a methodological error. However, since the series is composed of monotonically descending terms[3] with alternating sign, we can safely assume that the norm of ϵ_V is not larger than the first neglected term a_{nmax+1}. Using this simple method to estimate the methodological error it is possible to compute the series up to machine precision, i.e. until the final result is not changed including further terms in the sum. In conclusion: within machine precision we can eliminate the methodological error!

[3] at least starting from a given a_n!

Structure Chart — Evaluation of a Taylor series

```
x:= .....
n:=0
sumnew:=x
a:=x
    n:=n+1
    taylor:=sumnew
    a:= -x*x*(2*n-1)/n/(2*n+1) * a
    sumnew:=sumnew+a
sumnew=taylor
print: x, taylor
```

We have thus shown that all deviations of the numerical results from the exact one can be attributed to the accumulation of roundoff errors. We can immediately verify this, repeating the same calculation with single and double precision. The results of these tests for different values of x are given in the second row of Table 1.2.

The comparison of the results of the Taylor series expansion with other methods shows that the Taylor series approximates the value of erfc(x)quite well for small arguments x, but breaks down for large x.
In this range of parameters a different algorithm must be employed.

In this case, we can use a so-called *continued fraction* [11], which has the form

$$\mathrm{erfc}(x) = \frac{e^{-x^2}}{\sqrt{\pi}} \cdot \left(\frac{1}{x+} \frac{1/2}{x+} \frac{1}{x+} \frac{3/2}{x+} \cdots \right) \quad .$$

However, the numerical calculation of a continued fraction has the disadvantage that we have to decide how many terms of the fraction we want to retain before the beginning of the calculation. This means that if we want to increase the number of terms of the series to reduce the methodological error, we have to restart the calculation from scractch, and cannot simply add further terms to an 'old' calculation as in the case of Taylor series [4].

In the present example we have calculated the continued fraction for 31 and 61 terms, both with simple and double machine precision. Analysing the results (Table 1.2, third and fourth row), we can draw the following conclusions:

- Differences in the results with 31 and 61 terms → *methodological error.*

[4] Such a possibility in form of a better calculation of a continued fraction can be found for example in [9], P.135ff).

- Differences in the results with single vs double precision → *roundoff errors.*

Tab.1.2: Evaluation of the function erfc(x) through a Taylor series and a continued fraction. All results are given in half-exponential form with 7 digits for the mantissa. The exponent E indicates a calculation in simple precision, the exponent D a calculation in double precision.

x	Taylor series	Cont. Fract.	Cont. Fract.
	$\epsilon_V = 0$	31 Terms	61 Terms
0.01	0.9887166E+00	0.1261318E+02	0.9042203E+01
	0.9887166D+00	0.1261318D+02	0.9042202D+01
0.1	0.8875371E+00	0.1401417E+01	0.1131721E+01
	0.8875371D+00	0.1401417D+01	0.1131721D+01
0.5	0.4795001E+00	0.4802107E+00	0.4795305E+00
	0.4795001D+00	0.4802107D+00	0.4795305D+00
1.	0.1572992E+00	0.1572995E+00	0.1572992E+00
	0.1572992D+00	0.1572995D+00	0.1572992D+00
2.	0.4677685E-02	0.4677735E-02	0.4677735E-02
	0.4677735D-02	0.4677735D-02	0.4677735D-02
3.	0.2991446E-04	0.2209050E-04	0.2209050E-04
	0.2209050D-04	0.2209050D-04	0.2209050D-04
4.	0.2364931E-02	0.1541726E-07	0.1541726E-07
	0.1544033D-07	0.1541726D-07	0.1541726D-07
5.	0.4645361E+02	0.1537460E-11	0.1537460E-11
	0.5458862D-07	0.1537460D-11	0.1537460D-11

We can summarize the above results as follows:

- Taylor series: Methodological errors are zero in the whole range of x, roundoff errors grow rapidly with increasing x.

  The calculation with Taylor series gives results with at least 7 exact digits up to $\sim x = 1$.

- Continued Fraction: Methodological errors decrease with increasing x, the roundoff error is very small in the whole range of x considered.

1.3.4 Example of stable and unstable algorithms

Many numerical methods are based on *recursion formulas* of the type:

$$y_n = a \cdot y_{n-1} + b \cdot y_{n-2} \qquad n = 2, 3, \ldots \quad .$$

When employing such formulas, it is important to consider carefully the *stability* of the algorithm:

An algorithm is defined stable (or unstable), when the error with respect to the exact result at the n-th step of the calculation decreases (or increases) in the following steps (from [2], P.9).

A very instructive example in this context is the numerical calculation of *spherical Bessel functions* through a ”forward recursion”:

$$j_o(x) = \frac{\sin x}{x} \qquad j_1(x) = \frac{\sin x}{x^2} - \frac{\cos x}{x}$$

$$j_l(x) = \frac{2l-1}{x} \cdot j_{l-1}(x) - j_{l-2}(x) \qquad l = 2, 3, \ldots, lmax$$

The numerical evaluation of this formula obviously involves no methodological error; the only possible source of error are roundoff errors! To evaluate the magnitude of the roundoff error, we can compare the results calculated in single and double precision (see table 1.3).

Tab.1.3: Numerical calculation of the spherical Bessel functions of order 0-9 through forward recursion.

```
          single prec.                      double prec.
 x =  0.3000
 L=  0     0.9850674E+00             0     0.9850674D+00
     1     0.9910274E-01             1     0.9910289D-01
     2     0.5960023E-02             2     0.5961525D-02
     3     0.2309625E-03             3     0.2558598D-03
     4     -.5708978E-03             4     0.8536426D-05
     5     -.1735790E-01             5     0.2329701D-06
     6     -.6358853E+00             6     0.5811086D-08
     7     -.2753767E+02             7     0.1884359D-07
     8     -.1376248E+04             8     0.9363682D-06
     9     -.7795982E+05             9     0.5304202D-04
 x =  1.0000
 L=  0     0.8414710E+00             0     0.8414710D+00
     1     0.3011687E+00             1     0.3011687D+00
     2     0.6203508E-01             2     0.6203505D-01
     3     0.9006739E-02             3     0.9006581D-02
     4     0.1012087E-02             4     0.1011016D-02
     5     0.1020432E-03             5     0.9256116D-04
     6     0.1103878E-03             6     0.7156936D-05
     7     0.1332998E-02             7     0.4790142D-06
     8     0.1988459E-01             8     0.2827691D-07
     9     0.3367050E+00             9     0.1693277D-08
 x = 10.0000
 L=  0     -.5440211E-01             0     -.5440211D-01
     1     0.7846694E-01             1     0.7846694D-01
     2     0.7794219E-01             2     0.7794219D-01
     3     -.3949584E-01             3     -.3949584D-01
     4     -.1055893E+00             4     -.1055893D+00
     5     -.5553451E-01             5     -.5553451D-01
     6     0.4450132E-01             6     0.4450132D-01
     7     0.1133862E+00             7     0.1133862D+00
     8     0.1255780E+00             8     0.1255780D+00
     9     0.1000964E+00             9     0.1000964D+00
 x = 20.0000
 L=  0     0.4564726E-01             0     0.4564726D-01
     1     -.1812174E-01             1     -.1812174D-01
     2     -.4836553E-01             2     -.4836552D-01
     3     0.6030358E-02             3     0.6030359D-02
     4     0.5047615E-01             4     0.5047615D-01
     5     0.1668391E-01             5     0.1668391D-01
     6     -.4130000E-01             6     -.4130000D-01
     7     -.4352891E-01             7     -.4352891D-01
     8     0.8653319E-02             8     0.8653319D-02
     9     0.5088423E-01             9     0.5088423D-01
```

The results in Table 1.3 clearly show that this method is strongly unstable, in particular for small arguments x. This instability leads to a complete breakdown of the method for increasing order of the Bessel function.
On the other hand, the stability of the method sensibly improves for larger values of x!

In order to obtain meaningful results also for small values of the argument, we have to switch from the "forward recursion" algorithm we used so far to a "backward" recursion:

$$j_{L+1}(x) = 0 \qquad j_L(x) = \delta$$

$$j_l(x) = \frac{2l+3}{x} \cdot j_{l+1}(x) - j_{l+2}(x) \qquad l = L-1, L-2, \ldots, lmax, lmax-1, \ldots, 0$$

In these expression, δ is an arbitrary (small) number, which is the starting value for the backward recursion. L is a natural number > lmax.

Even though the starting value of the recursion is arbitrary, the values converge rapidly to a number series $\alpha \cdot j_l$ for decreasing l. The initially unknown constant α can be estimated normalizing the value for $l = 0$ to the quantity $\sin x/x$.

Due to the arbitrariness of δ, the results obtained in this way are affected by a methodological error, which is smaller, the bigger the starting index L.

To estimate ϵ_V, we have repeated the calculation twice, i.e. for $L = 18$ and $L = 27$; in both cases we have performed calculations both in single and double precision.

The results of these tests are summarised in Table 1.4.
It appears that the backward recursion represents a very stable algorithm for the whole range of x considered: the results with single and double precision differ by at most two units in the seventh digit of the mantissa. Furthermore, we see that the methodological error increases with increasing values of the argument x.

Summary of Tables 1.3 and 1.4:

- For not too large arguments x (for $lmax = 9$ up to $\sim x = 10.$) it is definitely better to employ the backward recursion, due to its higher stability.

- For the large-x range (for $lmax = 9$ and from $x = 10.$) it is preferrable to employ the forward recursion which is not affected by methodological error.

Tab.1.4: Numerical calculation of spherical Bessel function of order 0-9 with backwards recursion.

```
                  single precision                     double precision
             backwards(18)   backwards(27)       backwards(18)   backwards(27)
 x =  0.3000
 L=  0    0.9850674E+00   0.9850674E+00       0.9850674D+00   0.9850674D+00
     1    0.9910290E-01   0.9910290E-01       0.9910289D-01   0.9910289D-01
     2    0.5961526E-02   0.5961525E-02       0.5961525D-02   0.5961525D-02
     3    0.2558598E-03   0.2558598E-03       0.2558598D-03   0.2558598D-03
     4    0.8536426E-05   0.8536425E-05       0.8536426D-05   0.8536426D-05
     5    0.2329583E-06   0.2329583E-06       0.2329583D-06   0.2329583D-06
     6    0.5378445E-08   0.5378444E-08       0.5378444D-08   0.5378444D-08
     7    0.1076069E-09   0.1076069E-09       0.1076069D-09   0.1076069D-09
     8    0.1899475E-11   0.1899474E-11       0.1899474D-11   0.1899474D-11
     9    0.2999847E-13   0.2999847E-13       0.2999847D-13   0.2999847D-13
 x =  1.0000
 L=  0    0.8414710E+00   0.8414710E+00       0.8414710D+00   0.8414710D+00
     1    0.3011687E+00   0.3011687E+00       0.3011687D+00   0.3011687D+00
     2    0.6203505E-01   0.6203505E-01       0.6203505D-01   0.6203505D-01
     3    0.9006580E-02   0.9006579E-02       0.9006581D-02   0.9006581D-02
     4    0.1011016E-02   0.1011016E-02       0.1011016D-02   0.1011016D-02
     5    0.9256115E-04   0.9256114E-04       0.9256116D-04   0.9256116D-04
     6    0.7156936E-05   0.7156935E-05       0.7156936D-05   0.7156936D-05
     7    0.4790134E-06   0.4790133E-06       0.4790134D-06   0.4790134D-06
     8    0.2826499E-07   0.2826498E-07       0.2826499D-07   0.2826499D-07
     9    0.1491376E-08   0.1491376E-08       0.1491377D-08   0.1491377D-08
 x = 10.0000
 L=  0    -.5440211E-01   -.5440211E-01       -.5440211D-01   -.5440211D-01
     1    0.7846695E-01   0.7846695E-01       0.7846695D-01   0.7846694D-01
     2    0.7794219E-01   0.7794220E-01       0.7794220D-01   0.7794219D-01
     3    -.3949586E-01   -.3949586E-01       -.3949585D-01   -.3949584D-01
     4    -.1055893E+00   -.1055893E+00       -.1055893D+00   -.1055893D+00
     5    -.5553451E-01   -.5553451E-01       -.5553451D-01   -.5553451D-01
     6    0.4450133E-01   0.4450133E-01       0.4450133D-01   0.4450132D-01
     7    0.1133862E+00   0.1133862E+00       0.1133862D+00   0.1133862D+00
     8    0.1255780E+00   0.1255780E+00       0.1255780D+00   0.1255780D+00
     9    0.1000964E+00   0.1000964E+00       0.1000964D+00   0.1000964D+00
 x = 20.0000
 L=  0    0.4564727E-01   0.4564726E-01       0.4564726D-01   0.4564726D-01
     1    -.8801447E-01   -.1812270E-01       -.8801444D-01   -.1812269D-01
     2    -.5884944E-01   -.4836567E-01       -.5884943D-01   -.4836567D-01
     3    0.7330211E-01   0.6031280E-02       0.7330208D-01   0.6031273D-02
     4    0.8450518E-01   0.5047661E-01       0.8450516D-01   0.5047661D-01
     5    -.3527479E-01   0.1668320E-01       -.3527476D-01   0.1668320D-01
     6    -.1039063E+00   -.4130086E-01       -.1039063D+00   -.4130085D-01
     7    -.3226432E-01   -.4352875E-01       -.3226432D-01   -.4352875D-01
     8    0.7970807E-01   0.8654292E-02       0.7970804D-01   0.8654284D-02
     9    0.1000162E+00   0.5088490E-01       0.1000161D+00   0.5088490D-01
```

Chapter 2

Numerical methods for linear inhomogeneous sets of equations.

2.1 The basic problem

Let us consider the n equations

$$\begin{array}{c} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \cdot \\ \cdot \\ \cdot \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n \end{array} \tag{2.1}$$

where the a_{ij} and the b_i are real quantities. We further require that:

$$\mid b_1 \mid + \mid b_2 \mid + \cdots + \mid b_n \mid \neq 0$$

With these assumptions (2.1) constitutes a *real, linear, inhomogeneous set of equations (or linear system) of n-th order.* The quantities $x_1, \ldots, x_n$, which satisfy simultaneously all the above equations, are called the *solutions* of the system.

(2.1) is usually expressed in matrix form:

$$A \cdot \mathbf{x} = \mathbf{b} \tag{2.2}$$

The solution of this kind of systems is a central problem in numerical mathematics, since a wide range of numerical methods, such as methods for the numerical interpolation, least-square methods, differential methods a.s.o. can be reduced to the problem of solving a set of inhomogeneous linear equations.

From a theoretical point of view, the solution of (2.1) does not present any difficulty, as long as the determinat of the *matrix of coefficients A* does not vanish, *i.e.* as long as the problem is *non-singular*:

$$\det(A) \neq 0$$

In this case the problem can be solved with *Cramer's rule.* In practice, however, using this rule for $n \geq 4$ is very complicated; for several other reasons this procedure is not appropriate for use on a computer.

As in other fields of numerical mathematics, also for the present case there are two classes of methods:

- **Direct methods:**
  These do not include any methodological error and therefore always return the exact solution - except for rounding errors. Indeed, rounding errors can have severe effects in direct methods, which employ algorithms that are computationally expensive.

  As an example in the following we will illustrate the it elimination method of Gauss, in the formulation of *Doolittle and Crout (LU decomposition).*

- **Iterative methods**
  are often characterized by a particularly simple algorithm, stable with respect to rounding errors, and economical in terms of memory. However, there are also cases where iterative methods do not converge. Furthermore, also when they converge, they do not give the exact solution to the problem, but only an approximate one, which is also affected by a truncation error.

  As an example of iterative method for a linear set of equations, in this chapter we will discuss the *Gauss-Seidel method.*

2.2 Aim of the direct methods: transformation of the matrix of coefficients into a triangular matrix.

The aim of all direct methods is to reduce the original system:

$$A \cdot \mathbf{x} = \mathbf{b}$$

to an equivalent one with the same eigenvector.

$$U \cdot \mathbf{x} = \mathbf{y} \tag{2.3}$$

U is a so-called *triangular matrix* with the property:

$$U = [u_{ij}] \qquad \text{with} \qquad u_{ij} = 0 \qquad \text{for} \qquad i > j$$

The reason for this substitution is easy: systems of the type (2.3) are easy to solve through *back-substitution.* – see eq. (2.12) and (2.13).

2.3 Gauss' elimination principle in the formulation of Doolittle und Crout (LU decomposition).

The method of *Gaussian elimination* consists of two steps, i.e. the reduction of $A\mathbf{x} = \mathbf{b}$ to the equivalent system
$U\mathbf{x} = \mathbf{y}$ and the solution of this system through a back-substitution. This method exploits a proposition of linear algebra:
A linear set of equations remains unchanged, if one adds to one of its equations a linear combination of the other rows.

Doolittle and Crout have shown that the gaussian algorithm can be reformulated as follows: a real matrix A can be represented as the product of two real matrices L and U, i.e.

$$A = L \cdot U \tag{2.4}$$

where U is an *upper* triangular matrix:

$$U = \begin{pmatrix} u_{11} & u_{12} & .... & u_{1n} \\ 0 & u_{22} & .... & u_{1n} \\ . & & & . \\ . & & & . \\ . & & & . \\ 0 & 0 & .... & u_{nn} \end{pmatrix}$$

and L is a *lower* triangular matrix of the form

$$L = \begin{pmatrix} 1 & 0 & 0 & .... & 0 \\ m_{21} & 1 & .... & 0 & \\ m_{31} & m_{32} & 1 & .... & 0 \\ . & & & & . \\ . & & & & . \\ . & & & & . \\ m_{n1} & m_{n2} & m_{n3} & .... & 1 \end{pmatrix}$$

The decomposition (2.4) of a matrix A is usually called the *LU* decomposition of A.

Without derivation: some simple algebra on the LU decomposition.

The transformation is done column by column, i.e. $j = 1, 2, \ldots, n$:

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} m_{ik} u_{kj} \qquad i = 1, \ldots, j-1 \tag{2.5}$$

$$\gamma_{ij} = a_{ij} - \sum_{k=1}^{j-1} m_{ik} u_{kj} \qquad i = j, \ldots, n \tag{2.6}$$

$$u_{jj} = \gamma_{jj} \tag{2.7}$$

$$m_{ij} = \frac{\gamma_{ij}}{\gamma_{jj}} \qquad i = j+1, \ldots, n \tag{2.8}$$

Since the elements on the main diagonal of L are all equal to one we do not need to store the m_{jj} in memory; we can thus fit all the relevant coefficients in n x n matrix (the 'LU Matrix'):

$$\text{LU-Matrix}: \begin{pmatrix} u_{11} & u_{12} & u_{13} & \ldots\ldots & u_{1n} \\ m_{21} & u_{22} & u_{23} & \ldots\ldots & u_{2n} \\ m_{31} & m_{32} & u_{33} & \ldots\ldots & u_{3n} \\ . & & & & . \\ . & & & & . \\ . & & & & . \\ m_{n1} & m_{n2} & m_{n3} & \ldots\ldots & u_{nn} \end{pmatrix} \tag{2.9}$$

At this point, we can exploit the decomposition of A into the into the matrices L and U to calculate the solution. We have:

$$A \cdot \mathbf{x} \equiv L \cdot U \cdot \mathbf{x} = L \cdot (U \cdot \mathbf{x}) = \mathbf{b}$$

Setting $U \cdot \mathbf{x} \equiv \mathbf{y}$, we obtain the system $L \cdot \mathbf{y} = \mathbf{b}$. This system, due to the particular form of the matrix L (lower triangular matrix), is easy to solve through *forward substituion.*

$$y_1 = b_1 \tag{2.10}$$

$$y_i = b_i - \sum_{j=1}^{i-1} m_{ij} y_j \qquad i = 2, 3, \ldots, n \tag{2.11}$$

On the other hand, since U is an upper triangular matrix, it is easy also to obtain the auxiliary vector $\mathbf{y}$ of the system $U \cdot \mathbf{x} = \mathbf{y}$, this time employing a *backward substitution*:

$$x_n = \frac{y_n}{u_{nn}} \tag{2.12}$$

$$x_i = \frac{1}{u_{ii}} \left[y_i - \sum_{j=i+1}^{n} u_{ij} x_j \right] \qquad i = n-1, n-2, \ldots, 1 \tag{2.13}$$

If we want to employ the method of Doolittle and Crout is used in practice, we have to use two different programs. The first performs a LU-decomposition of the matrix of coefficients, the second calculates the solution using the formulas (2.10-2.13).

2.3.1 Demonstration of a memory-efficient LU-decomposition

using a generical 3x3-Matrix.

Let us consider:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

j = 1: first column

(2.6) i=1 $\gamma_{11} = a_{11}$
i=2 $\gamma_{21} = a_{21}$
i=3 $\gamma_{31} = a_{31}$

.

.

(2.7) $u_{11} = \gamma_{11}$

.

.

(2.8) i=2 $m_{21} = \gamma_{21}/\gamma_{11}$
i=3 $m_{31} = \gamma_{31}/\gamma_{11}$

Now, *instead of the initial coefficients of the matrix A*, which will not be used any more, we store only the coefficients u_{11}, m_{21} und m_{31}. The matrix becomes:

$$\begin{pmatrix} u_{11} & a_{12} & a_{13} \\ m_{21} & a_{22} & a_{23} \\ m_{31} & a_{32} & a_{33} \end{pmatrix}$$

j=2: second column

(2.5) i=1 $u_{12} = a_{12}$

.

.

(2.6) i=2 $\gamma_{22} = a_{22} - m_{21}u_{12}$
i=3 $\gamma_{32} = a_{32} - m_{31}u_{12}$

.

.

(2.7) $u_{22} = \gamma_{22}$

.

.

(2.8) i=3 $m_{32} = \gamma_{32}/\gamma_{22}$

$$\begin{pmatrix} u_{11} & u_{12} & a_{13} \\ m_{21} & u_{22} & a_{23} \\ m_{31} & m_{32} & a_{33} \end{pmatrix}$$

j=3: third column

(2.5) i=1 $$u_{13} = a_{13}$$

i=2 $$u_{23} = a_{23} - m_{21}u_{13}$$

.

.

(2.6)i=3 $$\gamma_{33} = a_{33} - m_{31}u_{13} - m_{32}u_{23}$$

.

.

(2.7) $$u_{33} = \gamma_{33}$$

$$\begin{pmatrix} u_{11} & u_{12} & u_{13} \\ m_{21} & u_{22} & u_{23} \\ m_{31} & m_{32} & u_{33} \end{pmatrix} = \text{LU-matrix} \quad [\text{s. Eq.}(2.9)]$$

It is clear that if we subsequently replace, column-by-column, the matrix of coefficients A with the LU-matrix *we do not lose any information!* This example also shows that the new u_{ij} and m_{ij} coefficients are always placed in new positions. In the program, if we want to save space (memory), we can thus avoid defining new variables for the elements a_{ij}, u_{ij}, γ_{ij} and m_{ij}, but we can save all these values in a single array $A(,)$. This is shown in the structure chart 3 (LUDCMP). This replacement allows to save much memory, but admittedly spoils the readability of the program.

2.3.2 Optimization of the rounding error through partial pivoting.

Let us suppose that we have at our disposal a program which solves an inhomogeneous set of equations through the LU-decomposition. The numerical results obtained through this program are not affected by methodological error (this is a direct method!). All deviations from the 'true' results, therefore, must come from rounding errors.

We give now a concrete example. Let us consider the following set of three equations:

$$\begin{aligned} x_1 + 5923181x_2 + 1608x_3 &= 5924790 \\ 5923181x_1 + 337116x_2 - 7x_3 &= 6260290 \\ 6114x_1 + 2x_2 + 9101372x_3 &= 9107488 \end{aligned} \tag{2.14}$$

The exact solution of this system is: $x_1 = x_2 = x_3 = 1$.

The numerical solution (FORTRAN, simple precision, 4 Byte per real number) gives the following LU-Matrix and the following eigenvector:

```
             0.1000000E+01   0.5923181E+07   0.1608000E+04
LU-Matrix:   0.5923181E+07   -.3508407E+14   -.9524475E+10
             0.6114000E+04    0.1032216E-02  0.9101371E+07


                             0.9398398E+00
    x:                       0.1000000E+01
                             0.9995983E+00
```

As we can see, the effect of the rounding error is considerable! (in particular on x_1).

If we now change the order of the equations in (2.14), which in principle should have no influence on the eigenvector, the same program returns:

```
                0.5923181E+07   0.3371160E+06   -.7000000E+01
  LU-Matrix:    0.1032216E-02   -.3459764E+03   0.9101372E+07
                0.1688282E-06   -.1712019E+05   0.1558172E+12

                                0.9999961E+00
     x:                         0.1000068E+01
                                0.1000000E+01
```

this time the rounding errors are much smaller. If we re-order the lines with the sequence 2/1/3 we obtain the correct eigenvector up to machine precision:

```
                0.5923181E+07   0.3371160E+06   -.7000000E+01
  LU-Matrix:    0.1688282E-06   0.5923181E+07   0.1608000E+04
                0.1032216E-02   -.5841058E-04   0.9101372E+07

                                0.1000000E+01
     x:                         0.1000000E+01
                                0.1000000E+01
```

This means that the order in which the equations appear affect the size of the rounding error and on the quality of the numerical solution in a decisive way!

What is the 'right' order? If we compare the m_{ij}'s in the LU-matrices, we immediately recognize, that the rounding error is the smallest *when the absolute values of the m_{ij}'s are as small as possible.*
In order to keep the m_{ij} as small as possible, the absolute values of γ_{jj}

which appear in equation (2.8) must be as large as possible. We can achieve this easily, if we determine the γ_{jj} with the largest absolute value through eq.(2.6) and exchange the row in which this element occurs with the $j-th$ row. Only after this do we evaluate equations (2.7) and (2.8).
This strategy is called a *LU-decomposition with partial pivoting.* In general, expecially for higher-order systems, there is no chance to get a correct result without using some kind of pivoting. There are however also sets of equations (see for example [2], page 56) for which the LU-decomposition is stable also without pivoting. This happens if the matrix of coefficients is *symmetrical, tridiagonal, cyclically tridiagonal, diagonal-dominant or generally positive definite.*

It is clear that with the strategy of partial pivoting we can also avoid the problems caused by the presence of a zero element on the diagonal γ_{jj} in Eq.(2.8). The only exception occurs if during the calculation all the $\gamma_{ij}, \quad i = j, \ldots, n$ are *simultaneously* zero. In that case the matrix of coefficients is *singular* and the system of equations cannot be solved with this method!

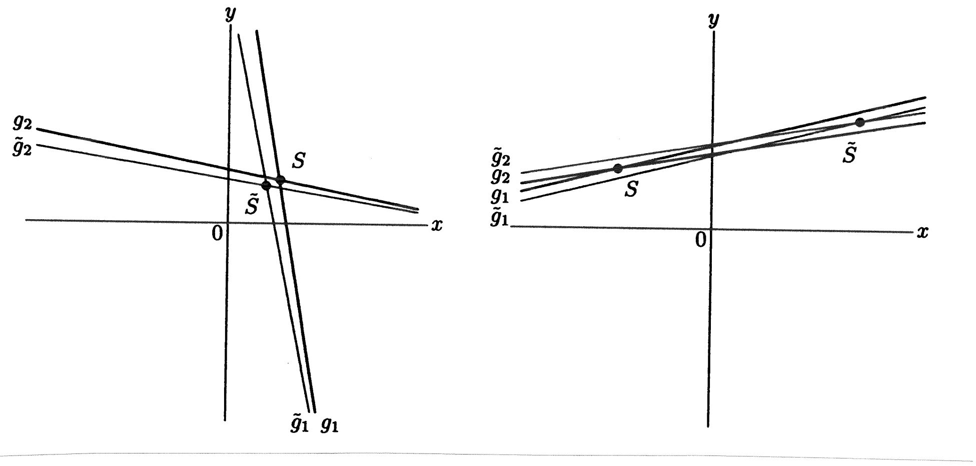


Figure 2.1: Good and ill-conditioned 2x2-System (from: [22], page 228).

2.3.3 Conditioning of a system of equations.

The solution of a linear set of equations calculated with a direct method most of the times does not coincide exactly with the true solution, because:

- During the calculation rounding errors can occur, and they can accumulate leading to wrong results.
- Uncertainities in the input values can cause also uncertainity in the solution.

If small variations in the output data cause large variations in the solution, one usually speaks of a *ill conditioned* system (see Fig.2.1).
The so-called *condition numbers* are practical ways to measure the quality of the solution of a given linear system.

In the mathematical literature a variety of condition numbers have been discussed. The program LUDCMP performs the calculation of the so called *condition number of Hadamard* for the matrix A, which is defined as: [2]

$$K_{\mathrm{H}}(A) = \frac{|det(A)|}{\alpha_1 \alpha_2 \cdots \alpha_n} \qquad \text{mit} \qquad \alpha_i = \sqrt{a_{i1}^2 + a_{i2}^2 + \ldots + a_{in}^2}.$$

For this number, the following empirical rules hold:

- $K_{\mathrm{H}}(A) < 0.01$ ill-conditioned,
- $K_{\mathrm{H}}(A) > 0.1$ well-conditioned,
- $0.01 \leq K_{\mathrm{H}}(A) \leq 0.1$ undefined.

2.3.4 The program LUDCMP.

Source: [9], p. 38ff; [10], p. 46ff (with small changes).

The program LUDCMP (LU DeCoMPosition) performs a LU-decomposition of a real square matrix.

INPUT Parameters:

A(,): Matrix of coefficients of the linear system.

N: Order of the system = number of rows and columns of A.

OUTPUT Parameters:

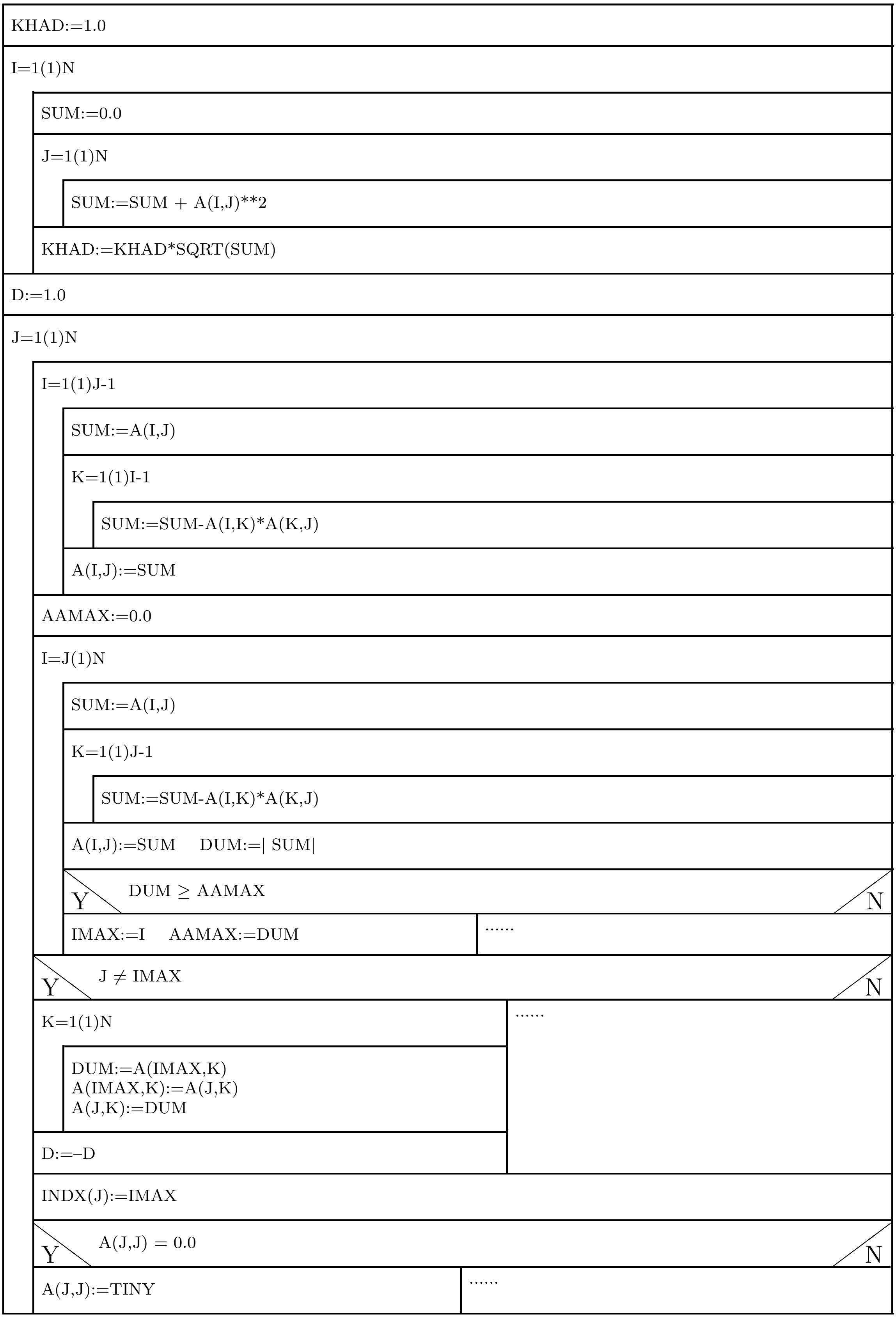
A(,): 'LU Matrix' of the original matrix A (see remark 1 after the structure chart).

INDX( ): Indexing vector, which stores the line replacements for the partial pivoting.

D: Determinant of the matrix of coefficients (see Section 2.3.6).

KHAD: Hadamard's condition number (See section 2.3.2).

Structure chart 3 — LUDCMP(A,N,INDX,D,KHAD)



```
KHAD:=1.0
I=1(1)N
    SUM:=0.0
    J=1(1)N
        SUM:=SUM + A(I,J)**2
    KHAD:=KHAD*SQRT(SUM)
D:=1.0
J=1(1)N
    I=1(1)J-1
        SUM:=A(I,J)
        K=1(1)I-1
            SUM:=SUM-A(I,K)*A(K,J)
        A(I,J):=SUM
    AAMAX:=0.0
    I=J(1)N
        SUM:=A(I,J)
        K=1(1)J-1
            SUM:=SUM-A(I,K)*A(K,J)
        A(I,J):=SUM     DUM:=| SUM|
        Y   DUM ≥ AAMAX   N
            Y: IMAX:=I     AAMAX:=DUM
            N: ......
    Y   J ≠ IMAX   N
        Y: K=1(1)N
               DUM:=A(IMAX,K)
               A(IMAX,K):=A(J,K)
               A(J,K):=DUM
           D:=–D
        N: ......
    INDX(J):=IMAX
    Y   A(J,J) = 0.0   N
        Y: A(J,J):=TINY
        N: ......
```

. — .

```
.
    DUM:=1.0/A(J,J)
    I=J+1(1)N
        A(I,J):=A(I,J)*DUM
    D:=D*A(J,J)
KHAD:=| D |/KHAD
(return)
```

Remarks on the program LUDCMP:

1. In order to save memory, the coefficients of the LU matrix are stored successively on the corresponding memory location of the original matrix A. This has however the disadvantage that the original matrix is lost. In case one wants to reuse the matrix A for other calculations, this has to be saved in another array **before** executing LUDCMP.

2. The output parameter D is used for the calculation of the determinant of A.

3. In [9], the constant TINY is set to 10^{-20}. As it is clear from the structure chart 3, this constant prevents the program from crashing in case one of the divisors of (2.8) becomes exactly zero. This is explained in section 2.5.

4. In [9] the matrix coefficients are scaled row-wise for the pivot search. The optimal scaling of a linear system represents a surprisingly complicated problem (see for example [7], page 140 ff and [12], page 41ff). These references also discuss the scaling of the program we have presented here.

2.3.5 The sub-program LUBKSB

Source: [9], page 39; [10], page 47 (with changes).

The program LUBKSB (LU BacKSuBstitution) calculates the solution $\mathbf{x}$ of the system $LU\mathbf{x} = \mathbf{b}$ using equations (2.10 - 2.13).

INPUT parameters:

A(,): LU-Matrix of the matrix of coefficients, calculated for example from program LUDCMP.

N: Order of the system = rows and columns of A.

INDX( ): Indexing vector, which contains information about the rows exchange performed in LUDCMP.

B( ): inhomogeneous vector of the system.

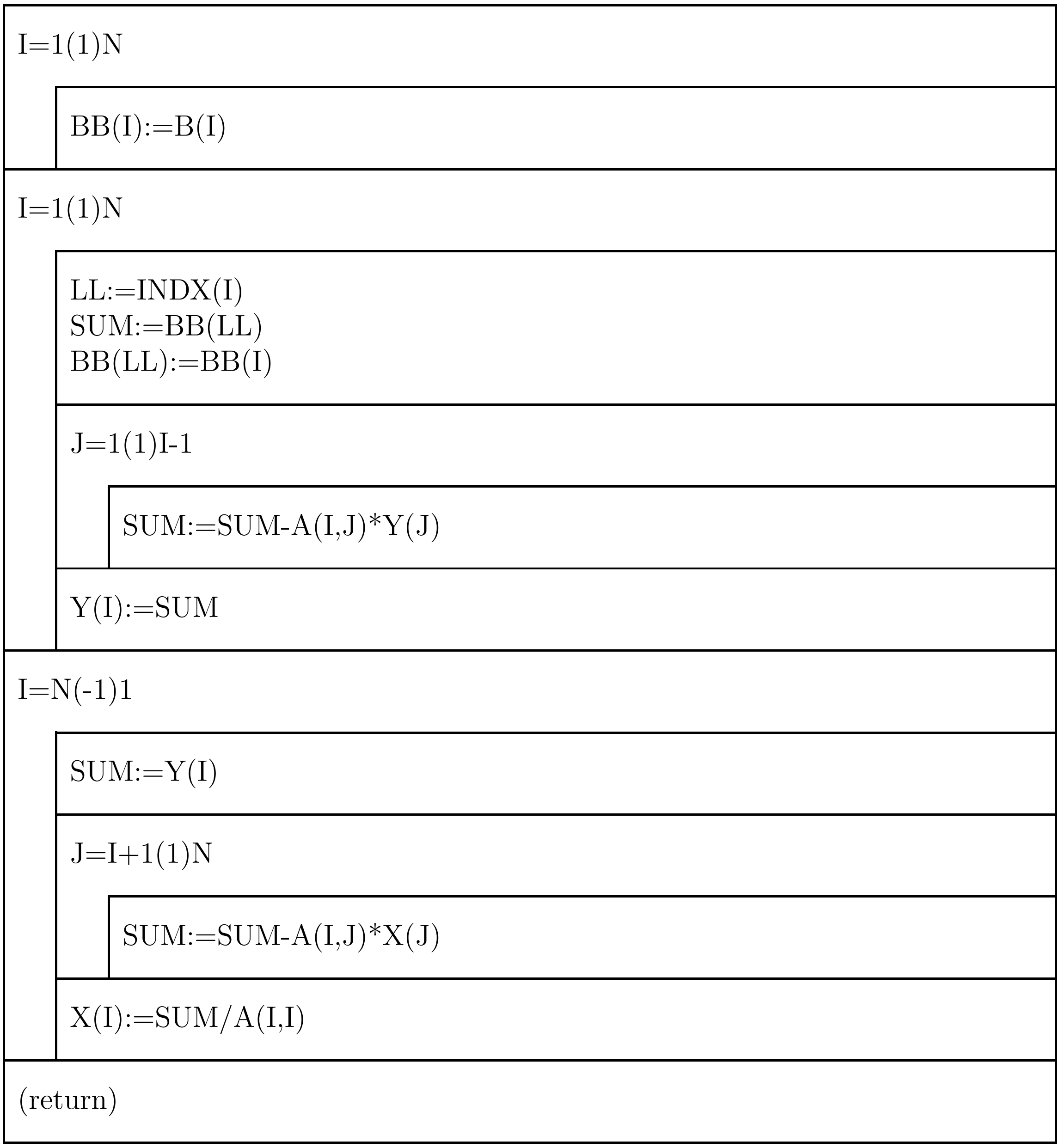
OUTPUT parameters:

X( ): Solution vector.

Internal vector:

Y( ): .

Structure chart 4 — LUBKSB(A,N,INDX,B,X)



```
I=1(1)N
    BB(I):=B(I)
I=1(1)N
    LL:=INDX(I)
    SUM:=BB(LL)
    BB(LL):=BB(I)
    J=1(1)I-1
        SUM:=SUM-A(I,J)*Y(J)
    Y(I):=SUM
I=N(-1)1
    SUM:=Y(I)
    J=I+1(1)N
        SUM:=SUM-A(I,J)*X(J)
    X(I):=SUM/A(I,I)
(return)
```

2.3.6 Possible uses of the programs LUDCMP and LUBKSB.

Solution of an inhomogeneous system $A\mathbf{x} = \mathbf{b}$:

```
||---------------------------||
||  LUDCMP(A,N,INDX,D,KHAD)  ||
||---------------------------||
||   LUBKSB(A,N,INDX,B,X)    ||
||---------------------------||
```

An advantage of the method we have just illustrated is that in case the inhomogeneous vectors $\mathbf{b}_1$, $\mathbf{b}_2$, ... are different, but the matrix of coefficients A is the same, it is sufficient to perform the LU decomposition only once:

```
||-------------------------||
|| LUDCMP(A,N,INDX,D,KHAD) ||
||-------------------------||
             |
||-------------------------||
|| LUBKSB(A,N,INDX,B1,X1)  ||
|| LUBKSB(A,N,INDX,B2,X2)  ||
||           .             ||
||           .             ||
||-------------------------||
```

Inversion of a matrix:

Using the two programs LUDCMP and LUBKSB a given matrix A can be inverted *column by column*:

$$A \cdot X = I \qquad I = \text{Identity matrix} \qquad X \equiv A^{-1}$$

This matrix equation can be rewritten as a series of inhomogeneous linear sets of equations of the type

$$A \begin{pmatrix} x_{11} \\ . \\ . \\ . \\ x_{n1} \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ . \\ . \\ 0 \end{pmatrix} \quad \cdots \quad A \begin{pmatrix} x_{1n} \\ . \\ . \\ . \\ x_{nn} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ . \\ . \\ 1 \end{pmatrix}$$

i.e. one has to solve n systems which have the same matrix of coefficients A and different inhomogeneous vectors (=columns of the identity matrix)

Structure chart — Inversion of a matrix

```
LUDCMP (A,N,INDX,D,KHAD)
J=1(1)N
    I=1(1)N
        VEKTOR(I):=0.0
    VEKTOR(J):=1.0
    LUBKSB (A,N,INDX,VEKTOR,X)
    I=1(1)N
        AINV(I,J):=X(I)
```

Calculation of the determinant of a matrix:

For the determinant of the matrix A we have

$$det(A) = det(L \cdot U) = det(L)\, det(U)\,.$$

Since both L and U are triangular matrices, their determinants are equal to the products of the elements on the diagonal. However, since all the diagonal elements of the L-Matrix have value 1, we also have

$$det(L) = 1\,,$$

from which it follows that

$$det(A) = det(U) = \sum_{i=1}^{n} u_{ii}\,.$$

In the program LUDCMP the calculation of $det(U)$ is performed during the LU-decomposition; we have to recall, however, that upon every exchange of rows the sign of the determinant will change.

2.3.7 Examples for the programs LUDCMP and LUBKSB.

Let us consider the square 4 × 4 matrix with (four rows and four columns):

$$A = \begin{pmatrix} 1.1161 & 0.1254 & 0.1397 & 0.1490 \\ 0.1582 & 1.1675 & 0.1768 & 0.1871 \\ 0.1968 & 0.2071 & 1.2168 & 0.2271 \\ 0.2368 & 0.2471 & 0.2568 & 1.2671 \end{pmatrix}$$

The condition number is $K_{\mathrm{H}} = 0.752$, i.e. the system is *well conditioned!*

- What is the solution $\mathbf{x}$ of the linear inhomogeneous system

  $$A \cdot \mathbf{x} = (-1.8367, 1.1944, 3.2368, -0.7232)^T \quad ?$$

  The numerical result is exact up to machine precision:

  $\mathbf{x} = (-.2000000E{+}01, 0.1000000E{+}01, 0.3000000E{+}01, -.1000000E{+}01)^T$

- What is the matrix inverse of A?

  The numerical result is exact almost up to machine precision (up to max two units in the seventh digit after the comma) and reads:

```
Inverted Matrix:

   .9379443E+00   -.6843720E-01   -.7960770E-01   -.8592076E-01
  -.8852433E-01    .9059826E+00   -.9919081E-01   -.1055899E+00
  -.1113511E+00   -.1169667E+00    .8784252E+00   -.1270733E+00
  -.1354557E+00   -.1401826E+00   -.1438075E+00    .8516058E+00

Test Matrix:

   .9999999E+00    .1126516E-07    .2518815E-08   -.1120211E-08
  -.4888310E-08    .1000000E+01   -.6621389E-08   -.3829147E-08
  -.2647183E-08    .5092004E-08    .9999999E+00    .1665558E-07
   .7356128E-08   -.1727934E-07   -.1442864E-07    .1000000E+01
```

  The test matrix contains the result of the multiplication: original matrix times inverted matrix. Ideally, this should be the identity matrix.

- What is the determinat of A?

  The numerical result (exact up to machine precision) is

  $$\text{Determinant} = 0.1758306E + 01$$

2.4 Iterative improvement of the solution.

It is clear that the accuracy of the numerical solution of a linear set of equations is limited by the machine precision. In many cases unfortunately we cannot attain this level of precision due to the accumulation of rounding errors.

In these cases we still have the possibility of *improving iteratively* the numerical solution. We will now briefly explain the (very simple) theory underlying this idea.

Let us suppose that the numerical solution of the system:

$$A \cdot \mathbf{x} = \mathbf{b}$$

is the vector $\bar{\mathbf{x}} = \mathbf{x} + \delta\mathbf{x}$, which is affected by a rounding error. If we now re-insert $\bar{\mathbf{x}}$ in the system of equations, we obtain a different inhomogeneous vector $\mathbf{b} + \delta\mathbf{b}$, due to the rounding error;

$$A \cdot \bar{\mathbf{x}} = A(\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b}$$

and

$$\underbrace{(A\mathbf{x} - \mathbf{b})}_{=0} + A \cdot \delta\mathbf{x} = \delta\mathbf{b}\,.$$

We have thus an error vector $\delta\mathbf{x}$, which is the solution of the system

$$A \cdot \delta\mathbf{x} = \delta\mathbf{b} \quad ,$$

where A is the *original* matrix of coefficients and $\delta\mathbf{b}$ is the so-called *residual vector* $A\bar{\mathbf{x}} - \mathbf{b}$.

The structure chart 5 illustrates how to solve a linear system with iterative improvement of the solution:

$$\mathbf{x} = \bar{\mathbf{x}} - \delta\mathbf{x} \quad .$$

A few more <u>remarks</u>:

1. Since the first time that LUDCMP is called the matrix A is destroyed, it is important to store the corresponding information somewhere before.

2. Due to the big risk of a *subtractive cancellation* the calculation of the residual vector should be performed in double precision, if possible.

3. Some software libraries (for example NAG) contain programs for the solution of inhomogeneous linear equations which already include a method for iterative improvement.

Structure chart 5 — Iterative improvement of the solution.

```
I=1(1)N
    J=1(1)N
        ASP(I,J):=A(I,J)
LUDCMP (A,N,INDX,D,KHAD)
LUBKSB (A,N,INDX,B,X)
    I=1(1)N
        SUMDP:=-B(I)
        J=1(1)N
            SUMDP:=SUMDP + DBLE(ASP(I,J))*DBLE(X(J))
        RES(I):=SUMDP
    LUBKSB (A,N,INDX,RES,DELX)
    I=1(1)N
        X(I):=X(I)-DELX(I)
(Condition, under which the iterative method should stop)
(Result and return)
```

2.5 Rounding errors in ill-conditioned and singular systems.

The method of partial pivoting permits a substantial reduction of the rounding errors which occur in the gaussian method. However, in *ill-conditioned* systems the numerical solution can still be seriously affected by errors, as we show in the following example:

```
0.10000E+01 0.50000E+00 0.33333E+00 0.25000E+00 0.20000E+00   0.228333E+01

0.50000E+00 0.33333E+00 0.25000E+00 0.20000E+00 0.16667E+00   0.145000E+01

0.33333E+00 0.25000E+00 0.20000E+00 0.16667E+00 0.14286E+00   0.109286E+01

0.25000E+00 0.20000E+00 0.16667E+00 0.14286E+00 0.12500E+00   0.884530E+00

0.20000E+00 0.16667E+00 0.14286E+00 0.12500E+00 0.11111E+00   0.745640E+00
```

The matrix of coefficients is a *Hilbert matrix* of fifth order, with components rounded at the fifth significant digit:

$$a_{ij} = \frac{1}{i+j-1}$$

and the inhomogeneous vector is chosen so that the exact solution is

$$x_1 = x_2 = x_3 = x_4 = x_5 = 1.$$

The condition number is $K_{\mathrm{H}} = 0.55 \cdot 10^{-10}$; i.e. this is a very ill-conditioned system, and we can expect to encounter many difficulties!

In the following table, we compare the exact and the numerical result $\bar{\mathbf{x}}$ (FORTRAN, single precision). We also show the vector $A \cdot \bar{\mathbf{x}}$:

$\mathbf{x}$ (exact)	$\bar{\mathbf{x}}$ (numerical)	$A \cdot \bar{\mathbf{x}}$
1.	0.9999459E+00	0.2283330E+01
1.	0.1000955E+01	0.1450000E+01
1.	0.9960009E+00	0.1092860E+01
1.	0.1005933E+01	0.8845300E+00
1.	0.9971328E+00	0.7456400E+00

The problem is clear: although some components of the numerical vector deviate considerably from the exact solution, the linear system of equations is still satisfied! Re-iteration doesn't improve the solution.

This example illustrates one of the most important problems in the use of numerical methods in linear systems. Despite several theoretical attempts (see for example [12] page 109 ff) there is no easy, general way to avoid that the components of the solution are affected by rounding errors.

Another unpleasant consequence of the rounding error is that in a numerical calculation it is practically impossible to discriminate between *singular* or *almost singular* problems. In the program a matrix is considered singular if all the elements of a pivot column are exactly zero. However, since these elements themselves are produced by an arithmetical operation, we could have elements that are not exactly zero due to rounding errors, *even if the matrix of coefficients is singular.* According to [12], page 63:

The situation is particularly unfortunate, because the strong mathematical difference between singular and non-singular matrices exists only in the ideal *world of mathematicians. As soon as we work with matrices in rounded arithmetics, the difference becomes necessarily much less clear-cut. Some non-singular matrices could become singular as a result of small perturbations introduced by rounding errors. It is even more likely that a genuine singular matrix is transformed into a similar, non-singular one.*

For this reason many programs avoid giving a definition of singularity and leave the choice to the user.
The program LUDCMP actually checks whether some diagonal elements γ_{jj} are exactly(!) zero, but only to avoid a division by zero in this (very unlikely) event!

2.6 Methods for direct solution of systems with special matrix of coefficients.

Since direct methods for linear systems are relatively expensive in terms of computational time and storage, many practical applications employ algorithms that are especially designed to exploit specific properties of the matrix of coefficients, if present.

2.6.1 Solution of systems of equations with tridiagonal matrices of coefficients.

Many important numerical methods (for example the spline interpolation) lead to linear sets of equations of the form

$$\begin{pmatrix} b_1 & c_1 & 0 & 0 & \cdots & 0 \\ a_2 & b_2 & c_2 & 0 & \cdots & 0 \\ 0 & a_3 & b_3 & c_3 & \cdots & 0 \\ . & & & & & . \\ . & & & & & . \\ . & & & & & c_{n-1} \\ 0 & 0 & 0 & 0 & \cdots & b_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ . \\ . \\ . \\ x_n \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \\ r_3 \\ . \\ . \\ . \\ r_n \end{pmatrix} \tag{2.15}$$

where the *tridiagonal* matrix of coefficients can clearly be rewritten in terms of the three vectors **b** (main diagonal), **a** and **c** (lower and upper secondary diagonal). In this case, if we use the LU-decomposition (without pivoting), we obtain the following structure:

$$\begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ m_2 & 1 & 0 & \cdots & 0 \\ 0 & m_3 & 1 & \cdots & 0 \\ . & & & & . \\ . & & & & . \\ . & & & & . \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix} \underbrace{\begin{pmatrix} u_1 & c_1 & 0 & \cdots & 0 \\ 0 & u_2 & c_2 & \cdots & 0 \\ . & & & & . \\ . & & & & . \\ . & & & & c_{n-1} \\ 0 & 0 & 0 & \cdots & u_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ . \\ . \\ . \\ x_n \end{pmatrix}}_{\equiv \mathbf{y}} = \begin{pmatrix} r_1 \\ r_2 \\ . \\ . \\ . \\ r_n \end{pmatrix}$$

which leads to the following simple expressions, through Eqs. (2.5-2.13):

$$\begin{aligned} u_1 &= b_1 \\ y_1 &= r_1 \end{aligned}$$

$$\begin{aligned} m_j &= a_j/u_{j-1} \\ u_j &= b_j - m_j \cdot c_{j-1} \qquad j = 2, \ldots, n \\ y_j &= r_j - m_j \cdot y_{j-1} \end{aligned} \tag{2.16}$$

The components of the solution can then be obtained through a back substitution:

$$\begin{aligned} x_n &= y_n/u_n \\ x_j &= (y_j - c_j \cdot x_{j+1})/u_j \qquad j = n-1, \ldots, 1 \end{aligned} \tag{2.17}$$

The following program exploits these expressions.

2.6.2 The program TRID.

Source: [9], p. 43 (with changes)

The program TRID calculates the solution of a linear, inhomogeneous set of equations with a tridiagonal matrix of coefficients.

INPUT parameters:

A( ),B( ),C( ): Vectors of the tridiagonal matrix.

R( ): Inhomogeneous vector.

N: Order of the system.

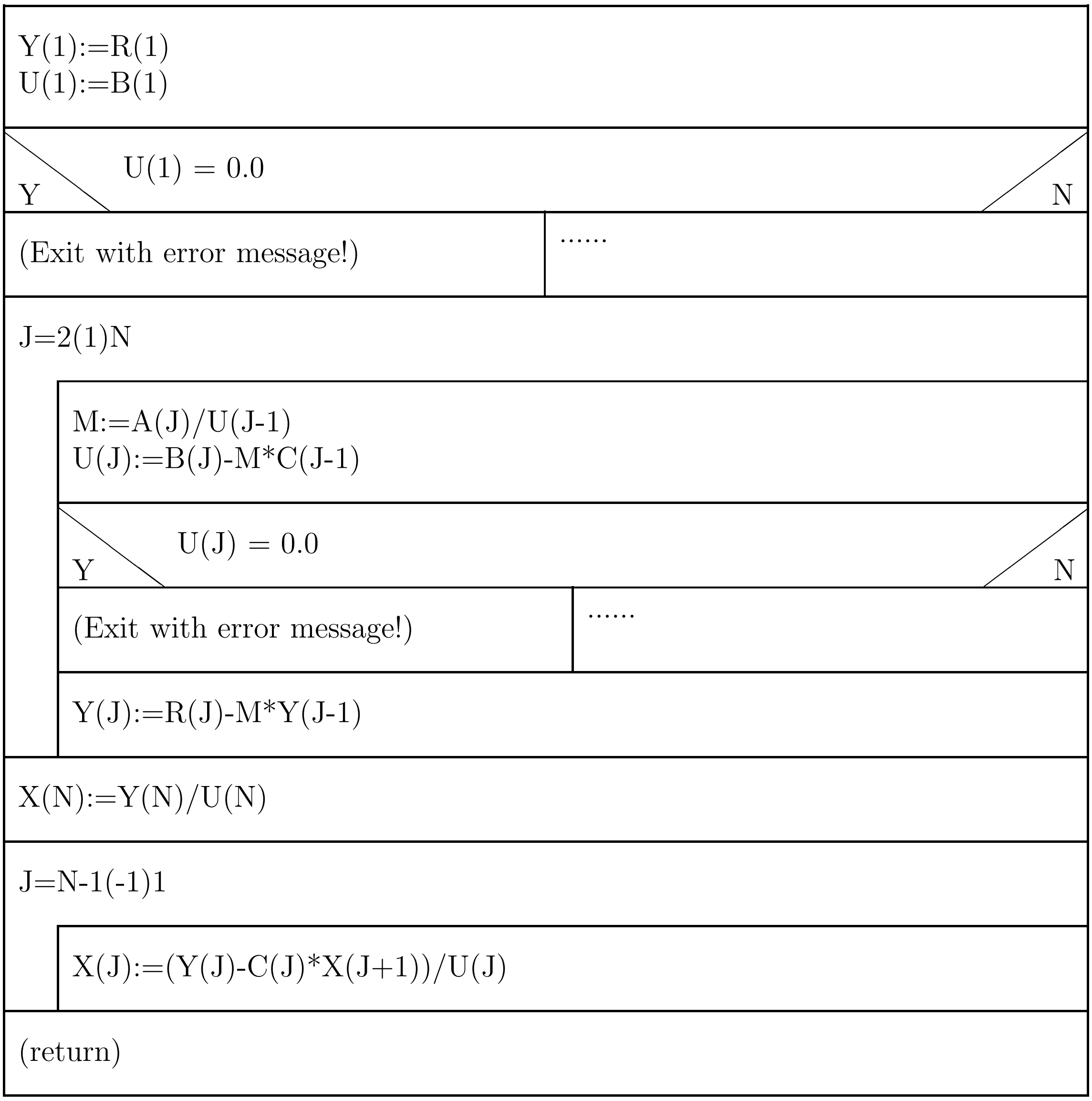
OUTPUT parameters:

X( ): Solution vector.

Internal vectors:

U( ), Y( ): .

Structure chart 6 — TRID(A,B,C,R,N,X)



```
Y(1):=R(1)
U(1):=B(1)
U(1) = 0.0
  Y: (Exit with error message!)
  N: ......
J=2(1)N
    M:=A(J)/U(J-1)
    U(J):=B(J)-M*C(J-1)
    U(J) = 0.0
      Y: (Exit with error message!)
      N: ......
    Y(J):=R(J)-M*Y(J-1)
X(N):=Y(N)/U(N)
J=N-1(-1)1
    X(J):=(Y(J)-C(J)*X(J+1))/U(J)
(return)
```

Remarks on the TRID program:

1. The main advantage of TRID is obviously a clear reduction of storage space, since as input information we have to provide only the four one-dimensional vectors A, B, C and R. The program also uses the four one-dimensional internal vectors U, Y, and the solution vector X. In a problem of n-th order this means that the storage space is of order $7n$. For comparison, the cost of a normal LU method is of order $n^2 + n$.

2. A further advantage compared to the normal LU method is the largely reduced computational cost.

3. On the other hand, not using pivoting means that the method may be aborted also in case of *non-singular* problems! In principle it would be possible to introduce pivoting also in the program TRID, but in this way the size of the matrix would increase (and in particularly unfavourable case even double). It can be shown, however, that this problem does not occur in the so-called *diagonal-dominant* tridiagonal matrices. These matrices have the property that $\mid b_j \mid > \mid a_j \mid + \mid c_j \mid$ for all $j = 1, \ldots, n$. This property is important, because many of the tridiagonal matrices that are encountered in practice are *diagonal-dominant*!

4. Many software libraries contain programs for the solution of linear sets of equations with tridiagonal matrices of coefficients: see for example [2], p. 304 and [9], p.42.

2.6.3 Other special forms of the matrix of coefficients.

In the literature we can find a large number of algorithms and programs designed to treat specific linear systems, whose matrix of coefficients have special structures. In particular, we refer the reader to [2] and [9], where he/she can find information on the following themes: symmetrical matrices (Cholesky-decomposition), cyclic-tridiagonal matrices, general band matrices, block matrices, ...

2.7 The Gauss-Seidel method.

2.7.1 General discussion.

The *Gauss-Seidel method*, together with its variants, known in the literature with the names *step-by-step method, Jacobi method, relaxation method, SOR-method, etc* is an *iterative method for the solution of linear inhomogeneous equations.*

As already mentioned at the beginning of this chapter, *iterative* methods offer some advantages with respect to direct methods, (simplicity of the algorithm etc), but also some disadvantages (possible convergence problems).

The main advantage of the iterative methods is however that *the elements of the original matrix are not changed during the iterative process.* As shown in the following, this is very convenient, if we have to solve linear systems of equations of higher order, which have *sparse matrix of coefficients.*

A matrix is called sparse if only a few of its elements are different from zero.

Since some very important problems of numerical mathematics (for example the solution of boundary values through a differential method) can be recast in terms of linear systems with sparse matrices, in the following we will mainly concentrate on these.

2.7.2 The basic principle of the Gauss-Seidel method.

We start once more with the linear system (2.1):

$$\begin{array}{c} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = f_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = f_2 \\ \cdot \\ \cdot \\ \cdot \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = f_n \end{array}$$

If we assume that *all* the elements on the diagonal of the matrix are non-zero, we can solve each of the equations with respect to the relative component of x on the diagonal:

$$x_1 = -\frac{1}{a_{11}}\left[a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n - f_1\right]$$

$$x_2 = -\frac{1}{a_{22}}\left[a_{21}x_1 + a_{23}x_3 + \cdots + a_{2n}x_n - f_2\right]$$

i.e. in general

$$x_i = -\frac{1}{a_{ii}}\left[\sum_{j=1(j\neq i)}^{n} a_{ij}x_j - f_i\right] \quad . \tag{2.18}$$

This reformulation can also be expressed in matrix form:

$$\mathbf{x} = C \cdot \mathbf{x} + \mathbf{b}$$

where

$$C = \{c_{ij}\} \quad \text{with} \quad c_{ij} = \begin{cases} -a_{ij}/a_{ii} & i \neq j \\ 0 & i = j \end{cases} \qquad \text{and} \qquad b_i = \frac{f_i}{a_{ii}}. \tag{2.19}$$

Rewriting (2.18) as:

$$x_i = x_i - \left(x_i + \frac{1}{a_{ii}} \left[\sum_{j=1 (j \neq i)}^{n} a_{ij} x_j - f_i \right] \right)$$

we immediately derive the *iteration rule for the Gauss-Seidel method.* If we insert into the right hand side of the equation above some approximate starting values for the x_i, we obtain *improved* approximate values for the components of the solution:

$$x_i^{(t+1)} = x_i^{(t)} - \Delta x_i^{(t)} \qquad (i = 1, \ldots, n) \tag{2.20}$$

with

$$\Delta x_i^t = x_i^{(t)} + \frac{1}{a_{ii}} \left[\sum_{j=1 (j \neq i)}^{n} a_{ij} x_j^{(t)} - f_i \right] \quad . \tag{2.21}$$

Obviously, we are assuming that the method converges. Through eq.(2.20) we obtain a *sequence of vectors*:

$$\mathbf{x}^{(0)} \quad ; \quad \mathbf{x}^{(1)} \quad ; \quad \mathbf{x}^{(2)} \quad ; \quad \ldots \quad ; \quad \mathbf{x}^{(t)}$$

that approximate the exact vector $\mathbf{x}$. We obviously never obtain the exact solution, but we can approach it as closely as possible (apart from rounding errors):

$$\lim_{t \to \infty} \mathbf{x}^{(t)} \to \mathbf{x}$$

The vector $\mathbf{x}^{(0)}$ is called the *starting vector* of the iteration.

Typically, for all iteration methods we have to specify the so-called *exit conditions* which determine when an iteration has to be stopped.

All iterative programs should contain at least the two following exit conditions:

- The iteration must stop when the results have reached the desired precision.
- The iteration should stop in any case, after a given maximum number of iteration steps ('emergency exit' in case of a divergence or very slow convergence).

2.7.3 The Gauss-Seidel method for band matrices.

A band matrix is a matrix that has non-zero elements only along some of its diagonals (see Fig.2.2, upper panel).

We immediately see that in terms of storage space it is a nonsense to store such a matrix as a (n x n)-matrix. The same amount of information can be

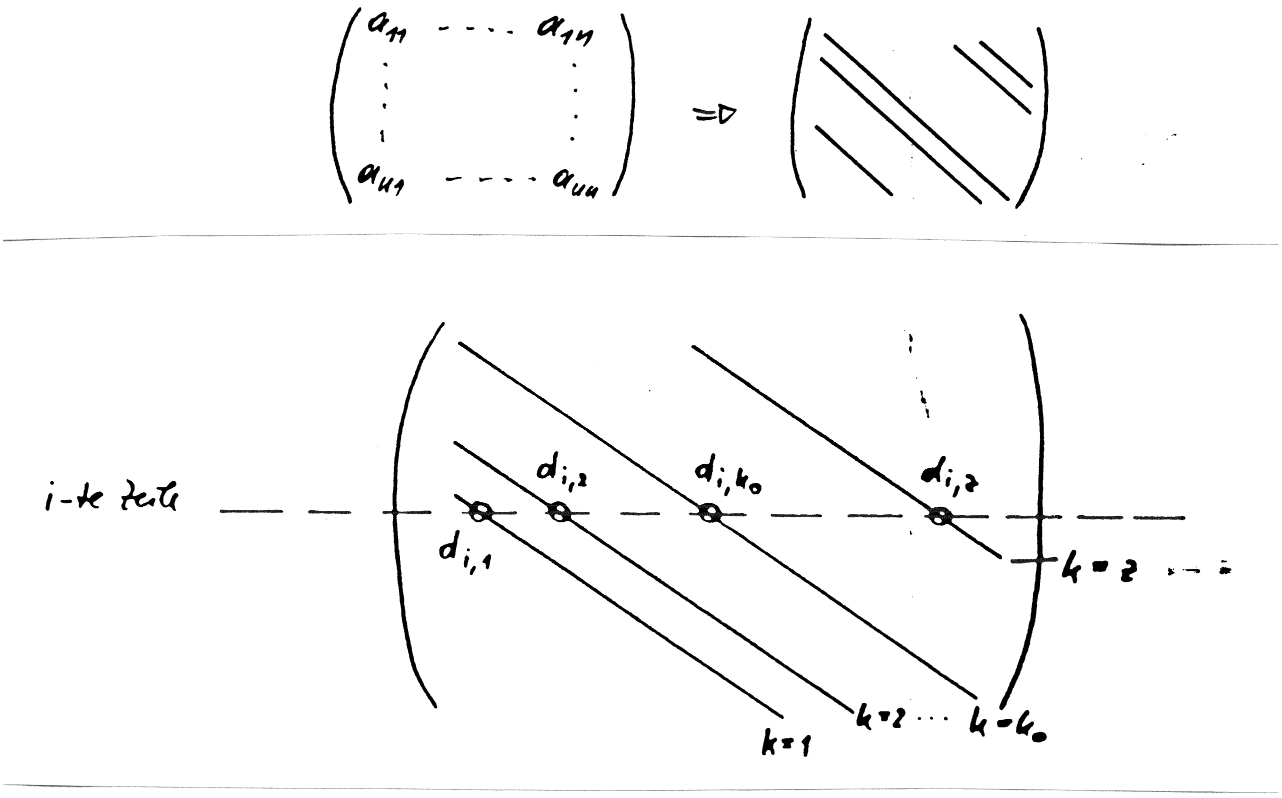


Figure 2.2: How to store a band matrix.

much more economically stored if we *save each of the non-zero diagonals as vectors, and additionally keep track of their positions in the original matrix.* This is illustrated schematically in the lower panel of Fig. 2.2:
Let us define z as the number of non-zero diagonals of the matrix. The position of each diagonal is represented by an integer number, which measures the position of this particular diagonal *relative to the main diagonal.* The whole information is thus stored in a two-dimensional array d_{ik}, where the index i indicates the matrix row. The second index (k) indicates which of the diagonals is considered. Furthermore, we also define an INTEGER vector $s(k)$ containing the relative positions:

$$s(k) = \begin{cases} \quad 0 & \text{Main diagonal} \\ +(-)1 & \text{first upper(lower) diagonal} \\ +(-)t & tt\text{–th upper(lower) diagonal} \end{cases}$$

Since in the Gauss-Seidel method the elements on the main diagonal play an expecially important role, the index $k = k_0$ of the main diagonal is particularly important. Therefore we set:

$$s(k = k_o) = 0$$

The matrix elements a_{ij} (i-th row, j-th column) are thus ordered into the elements of the two-dimensional array d_{ik} as:

$$a_{ij} = d_{ik} \quad \text{with} \quad j = s(k) + i \quad , \qquad (2.22)$$

where $i = 1, \ldots, n$ and $k = 1, \ldots, z$.

Usign these relations, Eq.(2.21) for the iteration method can be rewritten using (2.22):

$$x_i^{(t+1)} = x_i^{(t)} - \Delta x_i^{(t)} \tag{2.23}$$

with

$$\Delta x_i^{(t)} = x_i^{(t)} + \frac{1}{d_{i,k_o}} \left[\sum_{k=1 (k \neq k_o)}^{z} d_{ik} x_{s(k)+i}^{(t)} - f_i \right] \tag{2.24}$$

Note: The sum in (2.24) now contains only a few elements, for which:

$$0 < s(k) + i \leq n$$

2.7.4 Convergence criteria and error behaviour.

We are left now with the important question if it is possible to estabilish a priori whether a given linear system will be difficult to converge.

There is indeed in literature a long list of *convergence criteria*, i.e. of conditions which ensure convergence for the Gauss-Seidel iteration. All these criteria are *sufficient*, but not *necessary*. This means that the Gauss-Seidel method could converge, even in cases in which these criteria are not satisfied. For this reason here I will not list any mathematically-rigorous convergence criteria, but will limit myself to give a practical rule-of-thumb:

All linear systems in which the elements on the main diagonal dominate the other matrix elements have a good chance of convergence with the Gauss-Seidel method.

Obviously, convergence of the iteration means that the methodological error ϵ_V becomes smaller and smaller during the iteration, i.e.:

$$\lim_{t \to \infty} \epsilon_V^{(t)} \to 0$$

In practice, this means that we iterate the method until the correction vector $\Delta \mathbf{x}$ [Eq. (2.24)] becomes small enough. In this case, as exit criterion we can choose to monitor the *absolute* error:

$$\max \mid \Delta x_i \mid < \text{TOL}$$

or the *relative* error

$$\max \mid \frac{\Delta x_i}{x_i} \mid < \text{TOL}$$

where TOL is an error threshold chosen by the user of the program.

We have to be careful not to set a too low value for TOL. Indeed, if the problem is ill-conditioned, it can happen that we never reach the desired precision due to rounding errors!

Still, it is possible to say that iterative methods exhibit smaller rounding errors than direct methods. In fact, iteration methods have the desirable property that only the rounding error *of the last iteration* affects the results: there is no accumulation of ϵ_R through successive iterations! For this reason, in some cases the Gauss-Seidel method is preferrable to direct methods, if one wants to attain a higher precision.

2.7.5 The sub-program GAUSEI.

The program GAUSEI (GAUss-SEIdel) calculates the solution of an inhomogeneous linear set of equations with a general band matrix as coefficient matrix.

INPUT parameters:

N: Order of the system.

NDIAG: Number of non-zero diagonals in the band matrix.

S( ): INTEGER array containing the relative positions of the diagonals.

DIAG(,): Array with the matrix elements: the first index specifies the matrix row, the second the diagonal.

F( ): Inhomogeneous vector of the system.

TMAX: Maximum number of iteration steps.

W: Relaxation parameter (see section 2.7.6).

IREL: IREL $\neq$ 1: *absolute* error tolerance
IREL $=$ 1: *relative* error tolerance

TOL: Absolute or relative error which has to be reached during the iteration.

OUTPUT parameters:

SOL( ): Solution vector.

T: Number of iteration steps performed by GAUSEI.

ERROR: Logical variable for error diagnostic: After the execution of GAUSEI ERROR is 'false' if the required precision has been reached, and 'true' if

- not all the elements on the main diagonal are different from zero.
- the required precision has not been reached within TMAX iteration steps.

Important internal variables:

K0: Index of the main diagonal.

DX: Iteration-correction value according to Eq.(2.24).

ISCH: Control variable for the precision.

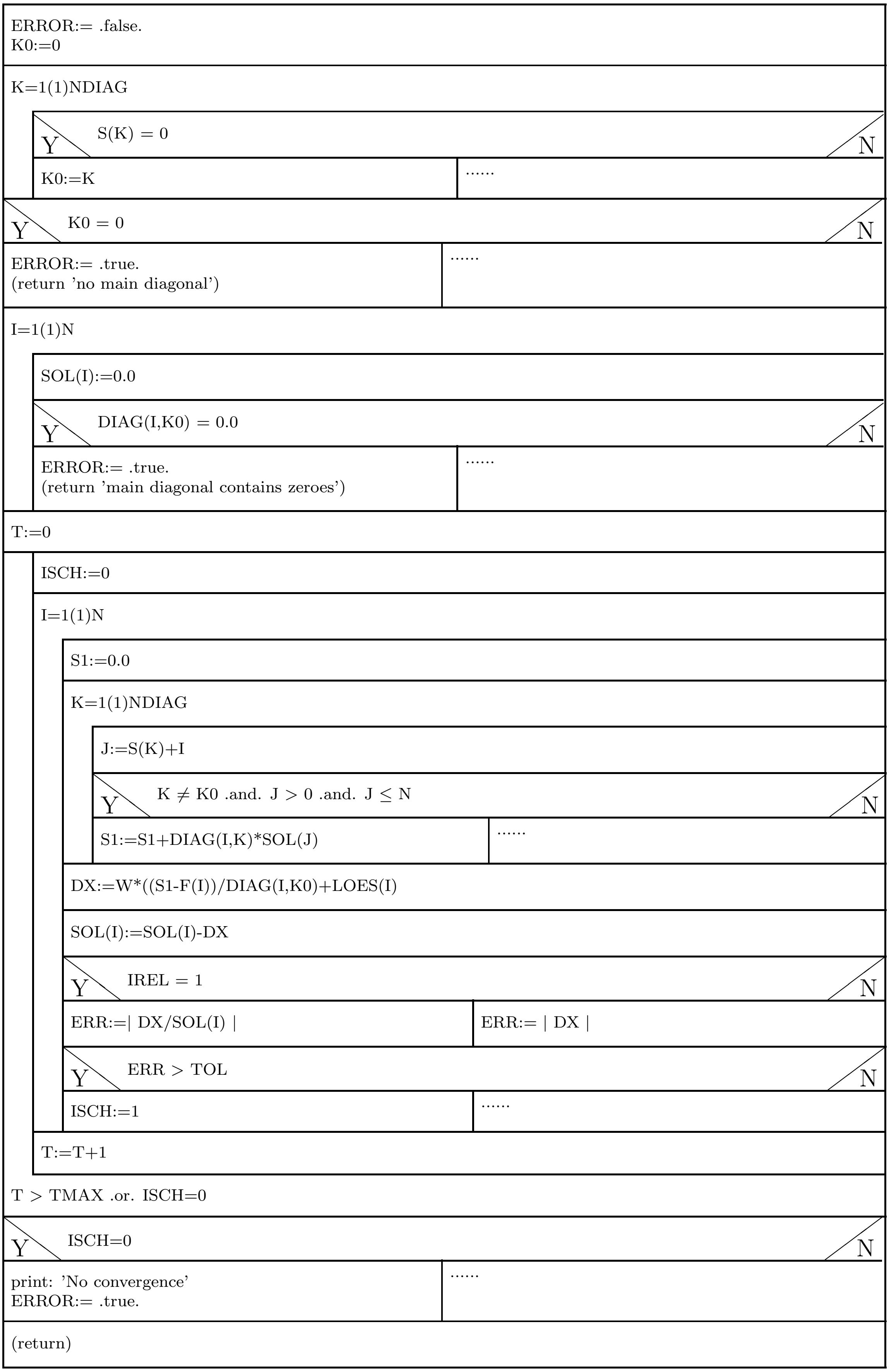
Structure of the program GAUSEI:

1. Check which of the given diagonals is the main diagonal. The corresponding index k is saved as K0. If none of the given diagonals is the main diagonal, the execution of the program is interrupted.

2. Check if the main diagonal contains zeroes. If yes, the program is interrupted. At this stage, the starting vector of the iteration (=zero vector) is defined.

3. The Gauss-Seidel iteration is performed with error monitoring.

Structure chart 8 — GAUSEI(N,NDIAG,S,DIAG,F,TMAX,W,IREL,TOL, SOL,T,ERROR)



```
ERROR:= .false.
K0:=0
K=1(1)NDIAG
    S(K) = 0
    Y: K0:=K
    N: ......
K0 = 0
Y: ERROR:= .true.
   (return 'no main diagonal')
N: ......
I=1(1)N
    SOL(I):=0.0
    DIAG(I,K0) = 0.0
    Y: ERROR:= .true.
       (return 'main diagonal contains zeroes')
    N: ......
T:=0
    ISCH:=0
    I=1(1)N
        S1:=0.0
        K=1(1)NDIAG
            J:=S(K)+I
            K ≠ K0 .and. J > 0 .and. J ≤ N
            Y: S1:=S1+DIAG(I,K)*SOL(J)
            N: ......
        DX:=W*((S1-F(I))/DIAG(I,K0)+LOES(I)
        SOL(I):=SOL(I)-DX
        IREL = 1
        Y: ERR:=| DX/SOL(I) |
        N: ERR:= | DX |
        ERR > TOL
        Y: ISCH:=1
        N: ......
    T:=T+1
T > TMAX .or. ISCH=0
ISCH=0
Y: print: 'No convergence'
   ERROR:= .true.
N: ......
(return)
```

2.7.6 A variant of the Gauss-Seidel method.

The iteration rule (2.23)

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \Delta\mathbf{x}^{(t)}$$

can be modified introducing a relaxation parameter ω:

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \omega \cdot \Delta\mathbf{x}^{(t)}. \tag{2.25}$$

If $\omega = 1$ we recover the normal Gauss-Seidel method, if $\omega > 1$ one speaks of a *overrelaxation method*, if $\omega < 1$ of an *underrelaxationmethod.*

Through a careful choice of the relaxation parameter ω in some cases the convergence speed can be sensibly enhanced. A good example of this effect can be found in [7], p. 192ff. If one iterates the simple linear system

$$\begin{aligned} x + 2y &= 3 \\ x - 4y &= -3 \end{aligned}$$

until the absolute error TOL=10^{-8} (C, double precision), one obtains the following relation between the number of convergence steps t and ω:

ω	t
0.65	20
0.70	18
0.75	15
0.8	14
0.85	12
0.9	12
0.95	21
1.0	31
1.05	48

This example shows that we could gain an enormous increase in efficience for the Gauss-Seidel method, if we could determine *a priori* the ideal value of the *relaxation parameter* ω_{ideal}. On this subject, we have to keep in mind the following remarks:

- The relaxation method converges <u>only</u> for $0 \leq \omega \leq 2$.
- 'Under specific mathematical restrictions', which apply to matrices that are used in differential methods for the solution of (mostly elliptical) boundary condition problems, ω_{ideal} is always between 1 and 2, i.e. there is always a overrelaxation.
- In these cases we have:

$$\omega_{\text{ideal}} = \frac{2}{1 + \sqrt{1 - \lambda_1^2}}\,, \tag{2.26}$$

where λ_1 is the largest eigenvalue of the matrix C [Eq.(2.19)].

- Since the exact calculation of λ_1 is often very demanding, I prefer to use an approximate method, described in Ref. [14], P. 63. This method behaves well in practice and is easy to incorporate in the program GAUSEI:

  It can be shown that the expression

  $$\frac{|\Delta \mathbf{x}^{(t)}|}{|\Delta \mathbf{x}^{(t-1)}|} \tag{2.27}$$

  tends to λ_1^2 for $t \to \infty$. If, for example in the program GAUSEI, we start the iteration with $\omega = 1$ and perform the first t_0 iterations [1], we can in many cases obtain a quite good approximation for λ_1^2 – Eq.(2.27), and thus for for ω_{ideal}, throguh Eq.(2.26). With this value of ω one can then continue the iteration.

In the exercises of this lecture we will demonstrate how well this strategy works.

2.7.7 Efficiency of the Gauss-Seidel method.

In the previous section we have mentioned that one of the advantages of the Gauss-Seidel method is that the matrix of coefficients is not changed during the iterations. This is particularly convenient in the case of sparse band matrices, where one has to store only the full diagonals.

We will now study a concrete example, namely the numerical solution of a very important partial differential equation, the Laplace equation. Using the differential method this differential equation and its constraints can be transformed in a set of linear equations. The order of this system is given by the number of **points in the main domain of the Laplace equation**. The following figure 2.3 (a) shows what percentage of the matrix elements of the linear system are different from zero as a function of the number of points in the main domain of the Laplace equation. We sees that, from 200 sampling points on, the percentage of matrix elements is well below 10 %. Fig. 2.3 (b) shows how this is translated in terms of memory.

Überhuber ([22], P. 392) gives an example of industrial application for linear methods (calculation of chassis rigidness in car industry): a matrix BC-SSTK32 has order 44.609, i.e. $2 \cdot 10^9$ elements; from these, however, 'only' 1.029.655 are non-zero, i.e. the occupation is 0.05 %.

[1] in practice a good value for t_0 is between 20 and 100

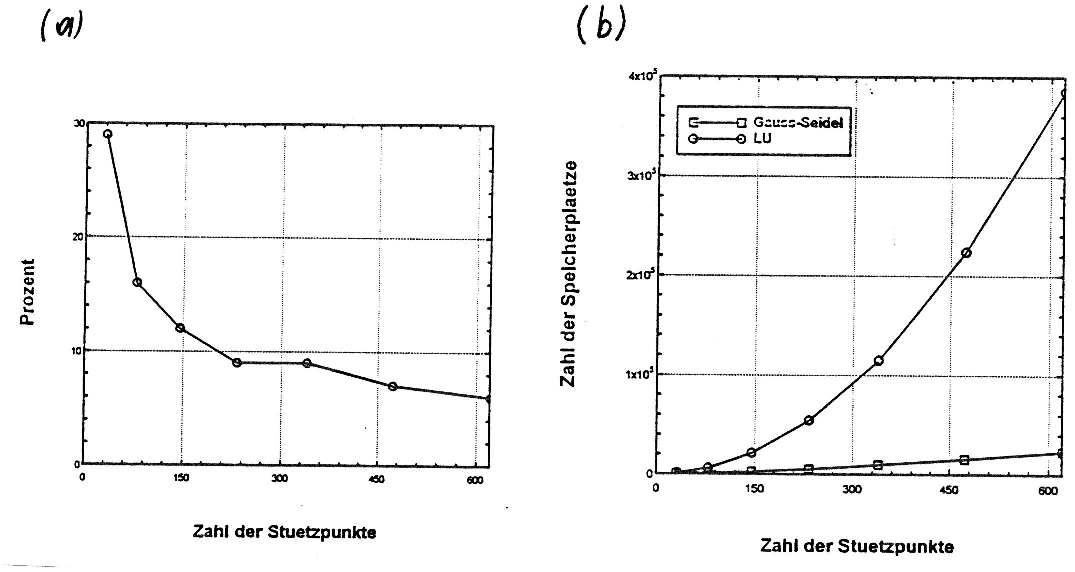


Figure 2.3: (a) Occupied matrix elements of the linear system (in %) as a function of number points in the main domain of the Laplace equation. (b) Storage space as a function of number of points, for Gauss-Seidel (squares) and LU (circles) methods.

Chapter 3

Least squares approximation

3.1 The basic problem.

Although the method of interpolation is very powerful in many cases, there are many other problems for which it is not possible to represent the data points with a smooth function. This happens in particular when the data points are *affected by statistical errors*, for example because they represent experimental data.

Let us imagine we measure the resistivity R of a metallic conductor using the experimental setup depicted in Fig.3.1. We obtain a *current-voltage diagram*, shown in Fig.3.2.

The relation between voltage and current is linear, and follows the *Ohm's law*:

$$U = R * I;$$

R is the *electrical resistivity.* Statistical deviations and possible measurement errors do not change the linear relation between U and I!

In this case, threfore, the approximating function must be a straight line (a polynomial of first order). Clearly, it doesn't make sense to approximate these data with an intepolation function that passes exactly through all measured points. In fact, what we rather wish to achieve is a *smoothening* of the data, without overweighing measurement errors.

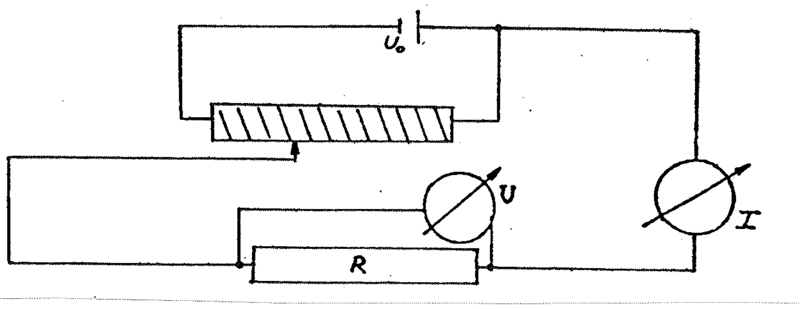


Figure 3.1: Experimental setup for the measurement of the electrical resistivity.

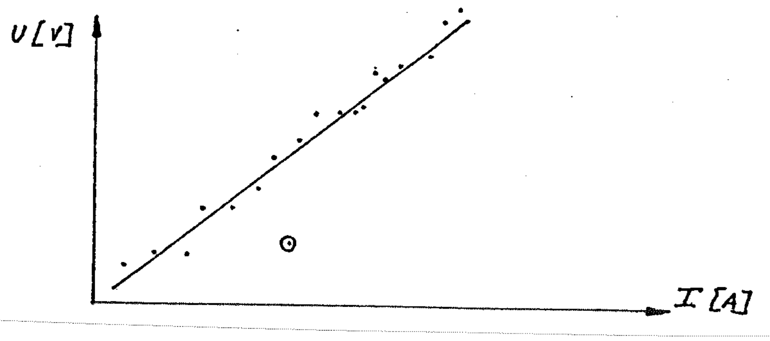


Abbildung 4.2: Ein Spannungs-Strom-Diagramm.

Figure 3.2: A current-voltage diagram.

3.2 Mathematical formulation of the problem.

Given a set of n points, $(x_i \mid y_i)$, we wish to find a curve $f(x)$ which approximates as closely as possible the points. At the same time, we want to take into account possible uncertainities due to measurement errors. We also wish to assign different weights to the points, through suitable weighting factors.

These requirements can be reformulated mathematically as follows – basic equation of the least squares (LSQ) approximation:

$$\chi^2 = \sum_{k=1}^{n} g_k \left[y_k - f(x_k; \mathbf{a})\right]^2 \quad \rightarrow \quad \text{minimum} \tag{3.1}$$

χ^2 is the *weighted error sum*, $g_k > 0$ is the *weighting factor* of the k-th point, and $f(x; \mathbf{a})$ is the *model function* with the q *model or fitting parameters*, $\mathbf{a} = a_1, a_2, \ldots, a_q$.

The sum of the weighted square errors between the "real" function and its approximate value should be minimal. The meaning of the weighting factor is clear, if we think in terms of statistical analysis of the data. We will explain this in more detail in the rest of the chapter.

3.3 Statistical analysis of the least squares problem.

3.3.1 Basic concepts: expectation value and standard deviation of a measure.

The LSQ method is based on the comparison between the experimental quantities y_k and the corresponding approximate values of the model function $f(x_k; \mathbf{a})$. It is obvious that the quality of the results (fir parameters) will depend strongly on the statistical quality of the y_k values.

Let us assume we repeat$n$ times a measure, under exactly the same conditions. The values of $x = x_k$ will correspond to (in general) *different* values

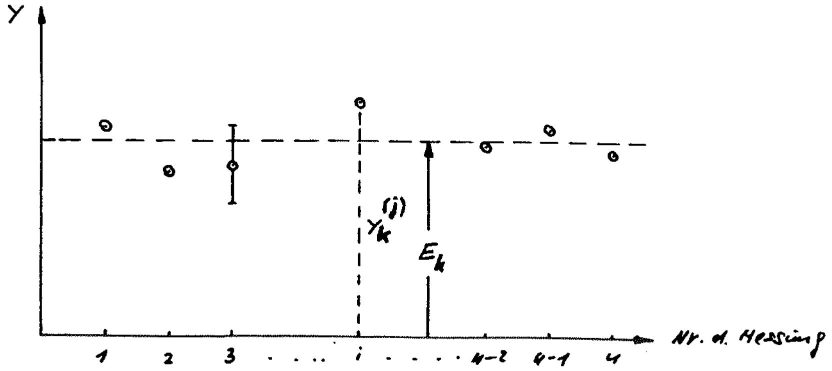


Figure 3.3: Expectation value of measured quantities.

$y_k^{(1)}, y_k^{(2)}, \ldots, y_k^{(n)}$. If we plot all these values on a diagram, as in Fig.3.3, we obtain a set of points distributed around the *expectation value* (or *mean value*) E_k:

$$E_k = \frac{1}{n} \sum_{j=1}^{n} y_k^{(j)} \tag{3.2}$$

A measure of the *spread* of the y_k values around their expaction value is the *standard deviation*:

$$\sigma_k = \left[\frac{\sum_{j=1}^{n} (y_k^{(j)} - E_k)^2}{n-1} \right]^{1/2} \tag{3.3}$$

In many practical cases the *probability* P with which a specific distance between the measured value and the expectation value occurs is given by the following *distribution*:

$$P(y - E) = \frac{1}{\sqrt{2\pi}\sigma} \cdot \exp^{-(y-E)^2/(2\sigma^2)} . \tag{3.4}$$

This is the very common *normal gaussian distribution* (Fig.3.4). It is important to stress in this case that $P(y - E)$ has inflection points exactly in $y - E = \pm\sigma$.
The shaded area in Fig.3.5 represents the probability that a measured value lies in the interval $[E - \sigma, E + \sigma]$. This probability is

$$\int_{-\sigma}^{+\sigma} d(y - E) \frac{1}{\sqrt{2\pi}\sigma} \exp^{(y-E)^2/(2\sigma^2)} = 0.685 \quad .$$

In a normal distribution, therefore, around 69 per cent ($\approx 2/3$) of all values lie in an interval $[E - \sigma, E + \sigma]$.

Normal distributions are typical of measurements, in which the measured quantity can assume all possible values within a given interval; these are called *analogue* measurements. In the next section we will discuss the *Poisson distribution*, which is instead typical of *digital* measurements.

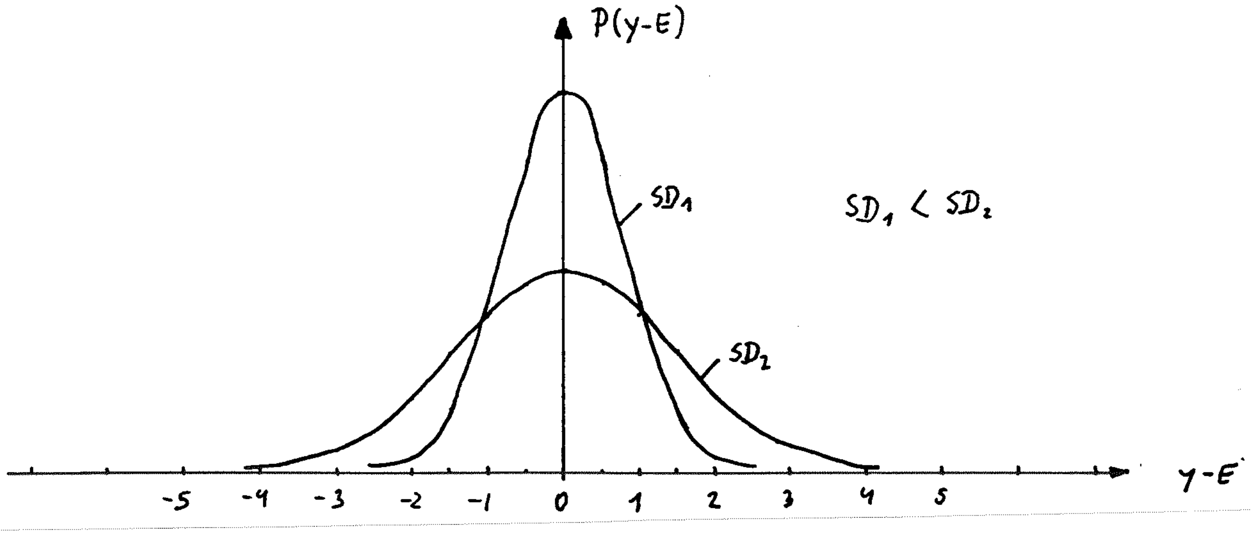


Figure 3.4: The normal gaussian distribution.

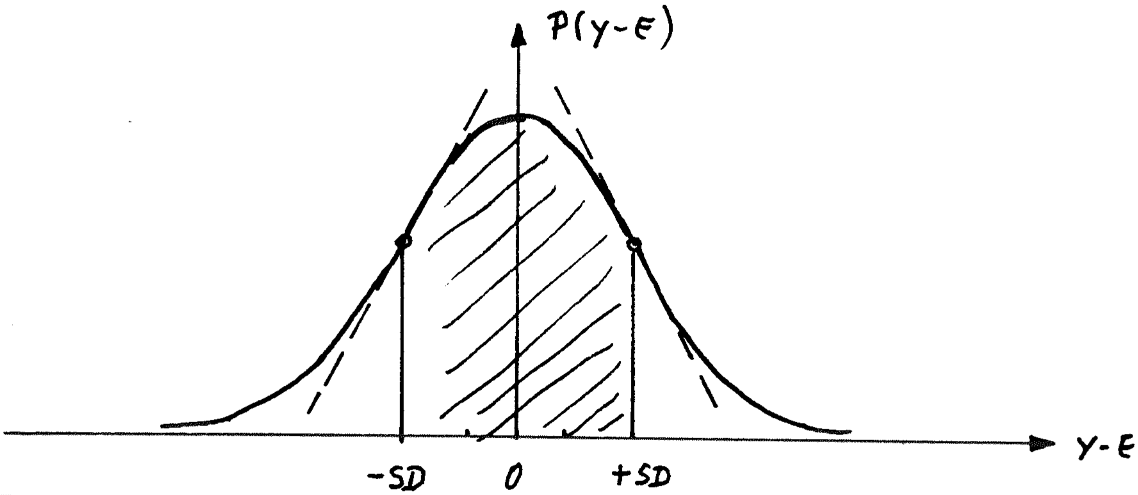


Figure 3.5: Geometrical meaning of the standard deviation σ.

3.3.2 Accounting for statistics in the LSQ method

Since this is not a course on statistic, we will limit ourselves to a few useful considerations, without proofs.

- In the LSQ method, he statistical uncertainity on the y_k's is taken into account assuming that the weighting factors in (3.1) are the squares of the reciprocal of the corresponding standard deviations:

$$g_k \equiv \frac{1}{\sigma_k^2} \quad . \tag{3.5}$$

- The LSQ method gives the statistical expectation value of the fitting parameters. The corresponding standard deviations can be obtained as follows:

  We first calculate the so-called *normal matrix* N, with coefficients:

$$[N]_{ij} = \frac{1}{2}\frac{\partial^2 \chi^2}{\partial a_i \partial a_j} \quad . \tag{3.6}$$

  Inverting the normal matrix we obtain the *covariance* matrix C:

$$C = N^{-1} \tag{3.7}$$

  The covariance matrix contains two important pieces of information:

  1. the diagonal elements are the standard deviations of the fitting parameters, i.e.
  $$\sigma_{a_i} = \sqrt{c_{ii}} \quad , \tag{3.8}$$
  2. the *correlation coefficients* of the fitting parameters (r_{ij}) can be extracted from C:
  $$r_{ij} = \frac{c_{ij}}{\sqrt{c_{ii}c_{jj}}} \quad . \tag{3.9}$$
     The r_{ij}'s, which always lie in the interval $[-1, +1]$, indicate how strongly the i-th and j-th fitting parameters infuence each other.

- Calculation of the *variance* V, or of the standard deviation:

$$V = \frac{\chi^2}{n-q} \quad \text{und} \quad \sigma_V = \sqrt{\frac{2}{n-q}} \quad . \tag{3.10}$$

  In case of an *ideal model* and for $n >> 1$, V has approssimatively a normal distribution with mean value 1 and standard deviation σ_V. The quantity $n - q$ (number of data points minus number of fitting parameters) is called the *number of degrees of freedom.* The variance can be used as an indicator for the quality of the model function employed in the fit: if V lies significantly outside the interval $[1 - \sigma_V, 1 + \sigma_V]$, the model is not good for the given set of points.

3.3.3 Determination of the standard deviations of the values.

The statistical evaluation of a LSQ problem requires us to know in the standard deviations σ_k of the measured values. How is it possible to calculate these σ_k's?

In practical cases there are two main possibilities:

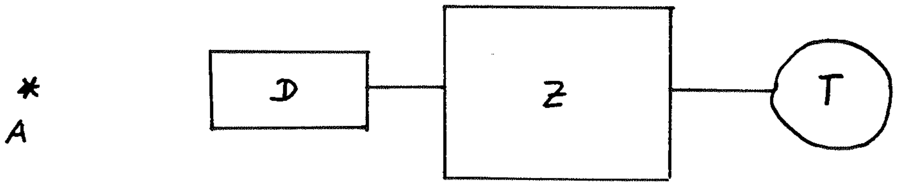
- The measured values have a normal distribution around their expectation value: this is a very common case in experiments. In this case it is a fairly good approximation to assume that the σ's *of all measured values are the same*, i.e. that

$$\sigma_1 \approx \sigma_2 \approx \cdots \approx \sigma_n .$$

  We can test this hypothesis, repeating several times the same measurement for a given x and calculating the corresponding σ's through (3.3).

- As already mentioned, there is another important statistical distribution, typical of counting experiments (digital measurements), i.e. the *Poisson* distribution.

Counting experiments play a major role, for example, in experimental physics. The experimental setup looks like this



where A is a radioactive substance, D a radiation detector, Z a digital counter and T a clock.

Without entering into the details of the Poisson statistics, we wish to illustrate here only its most important properties:

In a Poisson distribution the relation between the expectation value E of a measured quantity and the corresponding standard deviation is: σ

$$\sigma = \sqrt{E}.$$

In practice, the (unkwown) expectation value is approximated with the measured value, giving

$$\sigma \approx \sqrt{y} \quad .$$

In this case the weighting factors in the LSQ methods have the form:

$$g_k \approx \frac{1}{y_k} \quad . \tag{3.11}$$

3.4 Model Functions with Linear Parameters.

The term *model function with linear parameters* usually indicates a function of the form:

$$f(x; \mathbf{a}) = \sum_{j=1}^{m} a_j \cdot \varphi_j(x) \tag{3.12}$$

with arbitrary (linearly independent) *basis functions* $\varphi_j(x)$. The definition *linear model functions*, often used in practice, is misleading: $f(x; \mathbf{a})$ need not at all be linear in x, but only in the fitting parameters. In *linear regression* the model function is a straight line:

$$f(x; a_1, a_2) = a_1 + a_2\, x.$$

This is thus a very easy special case of model with linear parameters.

If we insert a function of the form (3.12) in the LSQ basic equation (3.1), we obtain:

$$\chi^2 = \sum_{k=1}^{n} g_k \left[y_k - \sum_{j=1}^{m} a_j \varphi_j(x_k) \right]^2 \quad \rightarrow \quad \text{minimum!}$$

The minimisation of the square error is obtained setting the partial derivatives of χ^2 with respect to the model parameters $\mathbf{a}$ to zero:

$$\frac{\partial\chi^2}{\partial a_i} = \sum_{k=1}^{n} g_k \varphi_i(x_k) \left[y_k - \sum_{j=1}^{m} a_j \varphi_j(x_k) \right] = 0 \quad i = 1, \ldots, m \quad .$$

This leads to a linear, inhomogeneous set of m-th order equations for the a_i's:

$$\sum_{j=1}^{m} a_j \sum_{k=1}^{n} g_k \varphi_i(x_k) \varphi_j(x_k) = \sum_{k=1}^{n} g_k y_k \varphi_i(x_k)$$

for all $i = 1, \ldots, m$, that is to the linear problem

$$A \cdot \mathbf{a} = \beta$$

with the symmetric, positive definite matrix of coefficient

$$A \equiv [\alpha_{ij}] \quad \text{with} \quad \alpha_{ij} = \sum_{k=1}^{n} g_k \varphi_i(x_k) \varphi_j(x_k) \tag{3.13}$$

and the inhomogeneous vector

$$\beta_i = \sum_{k=1}^{n} g_k y_k \varphi_i(x_k) \quad . \tag{3.14}$$

For linear model functions (3.12) the construction of the normal matrix according to (3.6) is very easy; we obtain in fact:

$$\frac{1}{2} \frac{\partial^2 \chi^2}{\partial a_i \partial a_j} = \sum_{k=1}^{n} g_k \varphi_i(x_k) \varphi_j(x_k) = \alpha_{ij} \quad . \tag{3.15}$$

3.4.1 Standard deviation of the fitting parameters

In section 3.3.2 we have shown (without proof) that the standard deviation of the fitting parameters can be derived from the diagonal elements of the covariance matrix [Eq. (3.8)]
We will now show that this is true using a very simple example, namely a *linear regression* with the model function:

$$f(x; a, b) = a + b\,x$$

Least-Squares Formula:

$$\chi^2 = \sum_{k=1}^{n} g_k \, [y_k - a - b\,x_k]^2 \quad \rightarrow \quad \text{Minimum for} \quad a = a_{opt}, \quad b = b_{opt}\,.$$

The normal matrix of the problem reads, according to (3.6):

$$N = \begin{pmatrix} \alpha_{11} & \alpha_{12} \\ \alpha_{12} & \alpha_{22} \end{pmatrix}$$

with

$$\alpha_{11} = \sum_{k=1}^{n} g_k \qquad \alpha_{12} = \sum_{k=1}^{n} g_k x_k \qquad \alpha_{22} = \sum_{k=1}^{n} g_k x_k^2$$

The optimised parameters can be obtained as solution of the inhomogeneous, linear system of equations:

$$N \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \beta_1 \\ \beta_2 \end{pmatrix}$$

with

$$\beta_1 = \sum_{k=1}^{n} g_k y_k \qquad \beta_2 = \sum_{k=1}^{n} g_k x_k y_k \quad .$$

The solution of this system of equations is

$$a = a_{opt} = \frac{\beta_1 \alpha_{22} - \beta_2 \alpha_{12}}{D} \qquad b = b_{opt} = \frac{\beta_2 \alpha_{11} - \beta_1 \alpha_{12}}{D} \tag{3.16}$$

where D is the determinant of the normal matrix

$$D = \alpha_{11}\alpha_{22} - \alpha_{12}^2 \quad .$$

The *covariance* C is the inverse of the normal matrix, i.e.

$$C = N^{-1} = \frac{1}{D} \begin{pmatrix} \alpha_{22} & -\alpha_{12} \\ -\alpha_{12} & \alpha_{11} \end{pmatrix}$$

From this one immediately obtains the *standard deviation of the fitting parameters* as

$$\sigma_a^2 = \frac{\alpha_{22}}{D} \qquad \sigma_b^2 = \frac{\alpha_{11}}{D} \, . \tag{3.17}$$

The calculation of this standard deviations can be obtained also in another way, using the error propagation rule (EPR).

It is clear that the optimised parameters a and b are functions of all the x and y components of the given points, i.e.

$$a = a(x_1, \ldots, x_n; y_1, \ldots, y_n) \qquad b = b(x_1, \ldots, x_n; y_1, \ldots, y_n)$$

According to the EPR we have for the standard deviations of a and b

$$\sigma_a^2 = \sum_{l=1}^{n} \left[\left(\frac{\partial a}{\partial x_l} \right)^2 \sigma(x_l)^2 + \left(\frac{\partial a}{\partial y_l} \right)^2 \sigma(y_l)^2 \right]$$

and

$$\sigma_b^2 = \sum_{l=1}^{n} \left[\left(\frac{\partial b}{\partial x_l} \right)^2 \sigma(x_l)^2 + \left(\frac{\partial b}{\partial y_l} \right)^2 \sigma(y_l)^2 \right]$$

The $\sigma(x_l)$ and $\sigma(y_l)$ are thus the standard deviations (errors) of the corresponding x and y values. If we assume that the x values are exact, $\sigma(x_l) = 0$ and for the errors on the y's we find, according to (3.5)

$$g_l = \frac{1}{\sigma(y_l)^2}$$

We thus obtain

$$\sigma_a^2 = \sum_{l=1}^{n} \frac{1}{g_l} \left(\frac{\partial a}{\partial y_l} \right)^2 \qquad \text{and} \qquad \sigma_b^2 = \sum_{l=1}^{n} \frac{1}{g_l} \left(\frac{\partial b}{\partial y_l} \right)^2$$

The partial derivatives in these equations can be calculated from (3.16), yielding:

$$\sigma_a^2 = \sum_{l=1}^{n} \frac{1}{g_l} \left[\frac{1}{D} \left(\alpha_{22} g_l - \alpha_{12} g_l x_l \right) \right]^2$$

and

$$\sigma_b^2 = \sum_{l=1}^{n} \frac{1}{g_l} \left[\frac{1}{D} \left(-\alpha_{12} g_l + \alpha_{11} g_l x_l \right) \right]^2$$

It is then trivial to show that:

$$\sigma_a^2 = \frac{1}{D^2} \sum_{l=1}^{n} g_l \left[\alpha_{22} - \alpha_{12} x_l \right]^2$$

$$= \frac{1}{D^2} \sum_{l=1}^{n} \sum_{k=1}^{n} \sum_{k'=1}^{n} g_l g_k g_{k'} \left[x_k^2 x_{k'}^2 - 2 x_l x_k^2 x_{k'} + x_l^2 x_k x_{k'} \right]$$

Renaming the indexes in the sum $l; k; k'$ we have thus

$$\sigma_a^2 = \frac{1}{D^2} \sum_{l,k,k'} g_l g_k g_{k'} \, x_k^2 \left[x_{k'}^2 - x_{k'} x_l \right] = \frac{1}{D^2} \sum_{k=1}^{n} g_k x_k^2 \left[\sum_{l=1}^{n} g_l \sum_{k'=1}^{n} g_{k'} x_{k'}^2 - \left(\sum_{l=1}^{n} g_l x_l \right)^2 \right]$$

The expression in the square bracket in the last line is clearly the determinant D of the normal matrix, and we obtain in the end

$$\sigma_a^2 = \frac{1}{D^2} D \sum_{k=1}^{n} g_k x_k^2 = \frac{\alpha_{22}}{D} \,,$$

which is the same expression we obtained using the covariance matrix (3.17), *quod erat expectandum.*

A very similar calculation can be done also for σ_b.

3.4.2 The program LFIT.

Source: [9], P.513ff, [10], P. 674ff with modifications.

INPUT parameters:

X( ), Y( ): Coordinates of the data points.

SIG( ): Standard deviations of the y's.

NDATA: Number of data points.

MA: Number of terms in the linear model function.

OUTPUT parameters:

A( ): Array with the optimised fitting parameters.

YF( ): Array with the values of the function of the best-fit curve on the given x points.

COVAR(,): Covariance matrix.

CHISQ: Weighted error sum χ^2.

Routines used by LFIT:

FUNCS: In this routine one defines the basis functions used in the model function (see Eq.(3.12)):

```
void funcs(double x, double afunc[], int ma)
{  ..... Definition of the local variables...
   afunc[1]=   .....  phi_{1}(x);
     .
     .
   afunc[ma] = .....  phi_{m}(x);
}
```

LUDCMP and LUBKSB: Routines for the solution of the linear system of equations [Eqs. (3.13),(3.14)] and for the inversion of the normal matrix, see chapter 2.

Structure of the program:

1. Calculation of the matrix elements α_{ij} and of the components β_i of the inhomogeneous vector according to (3.13) and (3.14). Storage of these quantities in NORMAL(,) and BETA( ). NORMAL contains the normal matrix of the problem, according to to eq.(3.15).

2. Calculation of the optimised fitting parameters and of the covariance matrix through the routines LUDCMP and LUBKSB.

3. Calculation of the function value of the best-fit curve for the given x coordinates and calculation of the minimal square error sum.

Structure chart 11 — LFIT(X,Y,SIG,NDATA,MA,A,YF,COVAR,CHISQ)

```
I=1(1)MA
    J=1(1)MA
        NORMAL(I,J):=0.0
    BETA(I):=0.0
K=1(1)NDATA
    FUNCS(X(K),AFUNC,MA)
    G:=1.0/SIG(K)/SIG(K)
    I=1(1)MA
        J=1(1)I
            NORMAL(I,J):=NORMAL(I,J) + G*AFUNC(I)*AFUNC(J)
        BETA(I):=BETA(I) + G*Y(K)*AFUNC(I)
I=2(1)MA
    J=1(1)I-1
        NORMAL(J,I):=NORMAL(I,J)
LUDCMP(NORMAL,MA,INDX,D,KHAD)
LUBKSB(NORMAL,MA,INDX,BETA,SOL)
J=1(1)MA
    A(J):=SOL(J)
CHISQ:=0.0
K=1(1)NDATA
    FUNCS(X(K),AFUNC,MA)
    G:=1.0/SIG(K)/SIG(K)
    SUM:=0.0
    J=1(1)MA
        SUM:=SUM + A(J)*AFUNC(J)
    YF(K):=SUM
    CHISQ:=CHISQ + G*(Y(K)-SUM)*(Y(K)-SUM)
J=1(1)MA
    I=1(1)MA
        BETA(I):=0.0
    BETA(J):=1.0
    LUBKSB(NORMAL,MA,INDX,BETA,SOL)
    I=1(1)MA
        COVAR(I,J):=SOL(I)
(return)
```

Remarks on LFIT:

1. The program requires to enter the standard deviations σ_k for the y values (array SIG). For this, as explained in Sect. 3.3.3. of this chapter, there are several possibilities:

   - Statistics of the data point unknown → all $\sigma_k = 1$.

     In this case it is obviously not possible to have any information about the variance of the fit or on the statistics of the optimised model parameters.
   - The data points have a normal distribution, with a standard deviation σ which is the same for all points ($\sigma_k \approx \sigma$).
   - The data points follow a Poisson distribution → $\sigma_k \approx \sqrt{y}_k$.

2. The program returns the optimal values of the fitting parameters, the value of the weighted error sum χ^2 and the covariance matrix. From this it is also possible to calculate the variance and standard deviation of the fit and the correlation coefficients of the fitted parameters, if needed.

3. We should also stress here that many of the LSQ programs available in literature (for example those contained in *Numerical Recipes*[9] and [10]) contain a a very useful option. When the LSQ program is called, the user can decide which of the MA parameters of the model function are to be optimised (fitting parameters), and which ones should instead remain unchanged during the whole LSQ process (fixed parameters).

3.4.3 Uses of LFIT.

The routine LFIT can be used for example within the following main program:

Program chart — :

```
Input: 1) The NDATA data points X() and Y() and SD SIG().
2) Number MA of the Model terms
LFIT(X,Y,SIG,NDATA,MA,A,YF,COVAR,CHISQ)
VAR:=CHISQ/(NDATA-MA)
I=1(1)MA
    SDPAR(I):=SQRT(COVAR(I,I))
I=1(1)MA-1
    J=I+1(1)MA
        NORM:=SQRT(COVAR(I,I)*COVAR(J,J))
        NORM ne 0.0
        Y:
            COVAR(I,J):=COVAR(I,J)/NORM
            COVAR(J,I):=COVAR(I,J)
        N:
            ......
I=1(1)MA
    COVAR(I,I) ne 0.0
    Y:
        COVAR(I,I):=1.0
    N:
        ......
Output: 1) the variance VAR.
2) The optimised parameters A().
3) The standard deviations of the parameters SDPAR().
4) The normalised covariance matrix COVAR(,).
5) optional: Table X() Y() YF()
(return)
```

An example.

In this example we will consider 101 data points normally distributed around their expectation value; we will assume that σ is the same for all points.

The expectation values of all points lie exactly on the third degree polynomial curve:

$$y(x) = 0.5 - x - 0.2x^2 + 0.1x^3 \quad .$$

The ideal model function for this problem is thus a third-degree polynomial with parameters: $a_1, \ldots, a_4$:

$$f(x; \mathbf{a}) = \sum_{j=1}^{4} a_j x^{j-1}$$

This means that the function FUNCS called by LFIT reads as follows (for example in C):

```
void funcs(double x, double afunc[], int ma)
{ int i;
  afunc[1]=1.0;
  for(i=2;i<=ma;i++) afunc[i]=afunc[i-1]*x;
}
```

As a first data set we consider a normal distribution with $\sigma = 0.1$.

Interpretation of Table 4.1 (a):

Depending on the number of terms in the model function, the number of degrees of freedom is between 96 and 100. For a good model, according to (3.10), we should expect a variance between 0.86 and 1.14 ca.

Due to the relatively large spread of the data points [see Fig.3.6 (a)] we cannot decide whether the best model is a second or third degree polynomial. In fact, for MA=4 the σ's of some of the fitting parameters are larger than their absolute values. These fitting parameters match very poorly the original expression, even though the 3^{rd} degree polynomial would be the correct fitting function.

$a_1^s = 0.5$	$a_1^{Fit} = 0.5097$
$a_2^s = -1.0$	$a_2^{Fit} = -1.1152$
$a_3^s = -0.2$	$a_3^{Fit} = -0.0517$
$a_4^s = 0.1$	$a_4^{Fit} = 0.0543$

The situation definitely improves if the data points have a better statistics. This can be seen it we consider a second data set with $\sigma = 0.025$.

Interpretation of Table 4.1 (b):

In this case it is much easier to decide which is the optimal model. The variance of the polynomials of zero, first and second order is too large, and the model functions can be considered good starting from MA$\leq$4. Since however for MA>4 the absolute values of some of the fitting parameters 'fall below their σ's', the third degree polynomial is the best model function. [see Fig.3.7 (b)].

Table 4.1 (a):

MA	Variance	optimised parameters	Remarks
1	37.034	$a_1 = -0.5652 \pm 0.0100$	bad model
2	1.138	$a_1 = 0.4574 \pm 0.0198$	border value
		$a_2 = -1.0226 \pm 0.0171$	
3	1.032	$a_1 = 0.5307 \pm 0.0293$	
		$a_2 = -1.2449 \pm 0.0676$	good model
		$a_3 = 0.1111 \pm 0.0327$	
4	1.035	$a_1 = 0.5097 \pm 0.0384$	
		$a_2 = -1.1152 \pm 0.1670$	good model,
		$a_3 = -0.0517 \pm 0.1945$	bad statistics
		$a_4 = 0.0543 \pm 0.0639$	s. Fig.3.6
5	1.046	$a_1 = 0.5134 \pm 0.0469$	
		$a_2 = -1.1543 \pm 0.3284$	good model
		$a_3 = 0.0372 \pm 0.6726$	bad statistics
		$a_4 = -0.0151 \pm 0.5065$	
		$a_5 = 0.0174 \pm 0.1256$	

Table 4.1 (b):

MA	Variance	optimized parameters	Remarks
1	598.559	$a_1 = -0.5639 \pm 0.0025$	bad model
2	2.883	$a_1 = 0.4773 \pm 0.0049$	bad model
		$a_2 = -1.0413 \pm 0.0043$	
3	1.204	$a_1 = 0.5472 \pm 0.0073$	
		$a_2 = -1.2530 \pm 0.0169$	bad model
		$a_3 = 0.1059 \pm 0.0082$	
4	0.899	$a_1 = 0.5128 \pm 0.0096$	good model
		$a_2 = -1.0413 \pm 0.0417$	the fitting parameters have
		$a_3 = -0.1600 \pm 0.0486$	convenient σ's.
		$a_4 = 0.0886 \pm 0.0160$	see Fig.3.6
5	0.908	$a_1 = 0.5141 \pm 0.0117$	good model
		$a_2 = -1.0548 \pm 0.0821$	but some fitting parameters
		$a_3 = -0.1294 \pm 0.1681$	have a very bad statistics.
		$a_4 = 0.0647 \pm 0.1266$	'mixing of parameters'
		$a_5 = 0.0060 \pm 0.0314$	

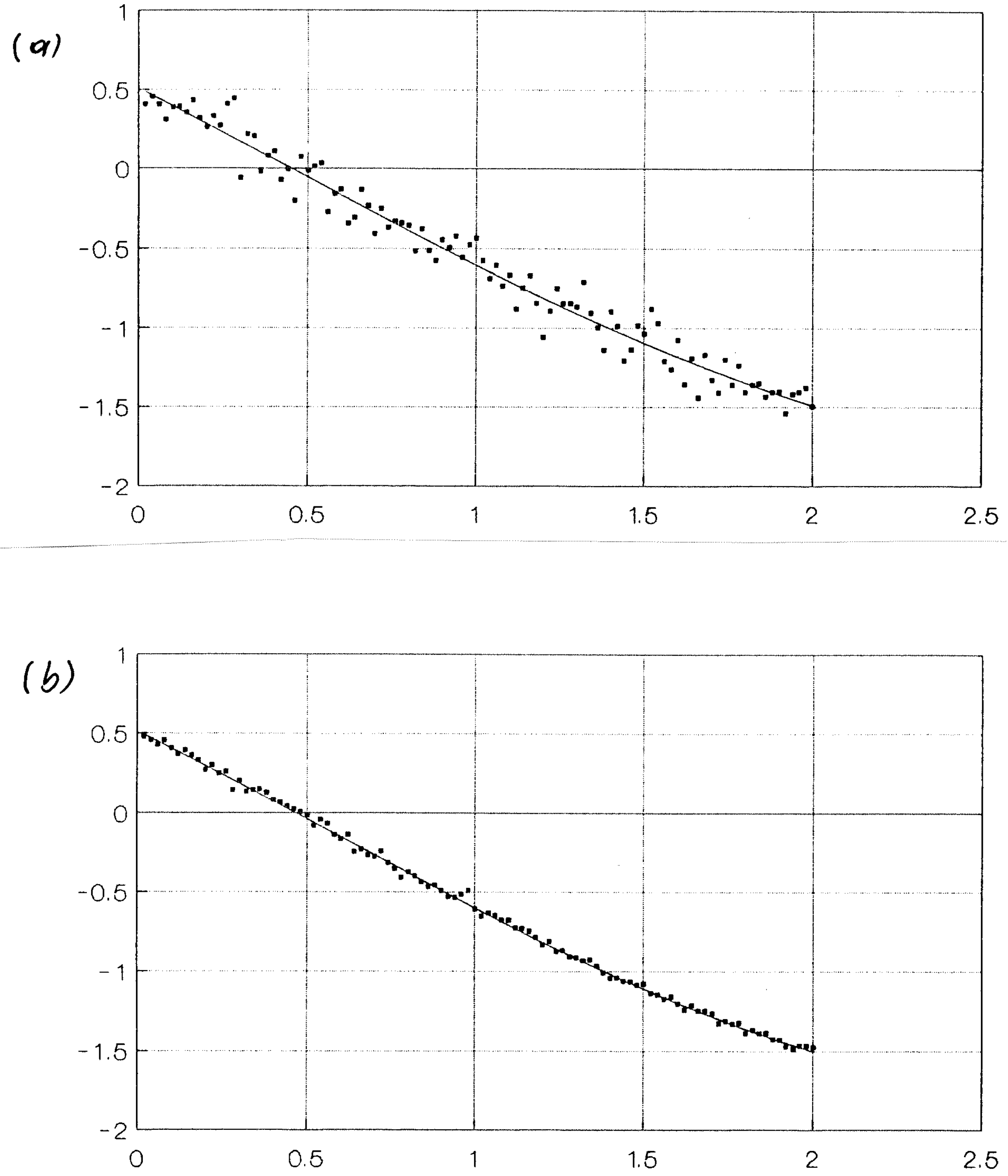


Figure 3.6: Linear LSQ calculation of simulated data values: (a) $\sigma = 0.1$, (b) $\sigma = 0.025$.

3.5 Model functions with non linear parameters.

3.5.1 What are non linear parameters?

If we insert the model function

$$f(x; a, b) = a \cdot e^{-bx}$$

in the LSQ equation (3.1), we obtain

$$\chi^2 = \sum_{k=1}^{n} g_k \left[y_k - a \cdot e^{-bx_k} \right]^2 \quad \rightarrow \quad \text{Minimum!}$$

If we now proceed as usual, and differentiate with respect to the parameter a, we obtain the linear equation

$$a \cdot \sum_{k=1}^{n} g_k e^{-2bx_k} = \sum_{k=1}^{n} g_k y_k e^{-bx_k} \quad .$$

A derivation with respect to b leads to the non-linear equation

$$a \cdot \sum_{k=1}^{n} g_k x_k e^{-2bx_k} = \sum_{k=1}^{n} g_k x_k y_k e^{-bx_k} \quad .$$

Therefore a and b are called the *linear* and *non linear* parameters of the model function $f(x)$, respectively. From this simple example it is easy to understand that we cannot apply the method described in section 3.4 to non linear model functions. A possibility to treat non-linear models within the LSQ approximation is represented by the *formalism of Gauss and Newton* treated in section 3.5.3.

Before explaining this, we will however show how to easily linearise specific non-linear model functions.

3.5.2 Linearization of non-linear problems.

A type of distribution often encountered in practice is the one shown in Fig.3.7. The $(x_k \mid y_k)$ values clearly obey an *exponential law.* Their distribution on a semi-logarithimc
$(x \mid \ln y)$ coordinate system is therefore approximately *linear.*

To fit points with this distribution it is common to use the model function:

$$f(x; a, \lambda) = a \cdot e^{-\lambda x},$$

which on a semi-logarithmic scale has the linear form

$$\ln y = \ln a - \lambda x.$$

The linear system of equations for the two fitting parameters reads, according to Eq. (3.13) and (3.14):

$$\begin{pmatrix} n & \sum_k x_k \\ \sum_k x_k & \sum_k x_k^2 \end{pmatrix} \cdot \begin{pmatrix} \ln a \\ -\lambda \end{pmatrix} = \begin{pmatrix} \sum_k \ln y_k \\ \sum_k x_k \ln y_k \end{pmatrix} \quad ,$$

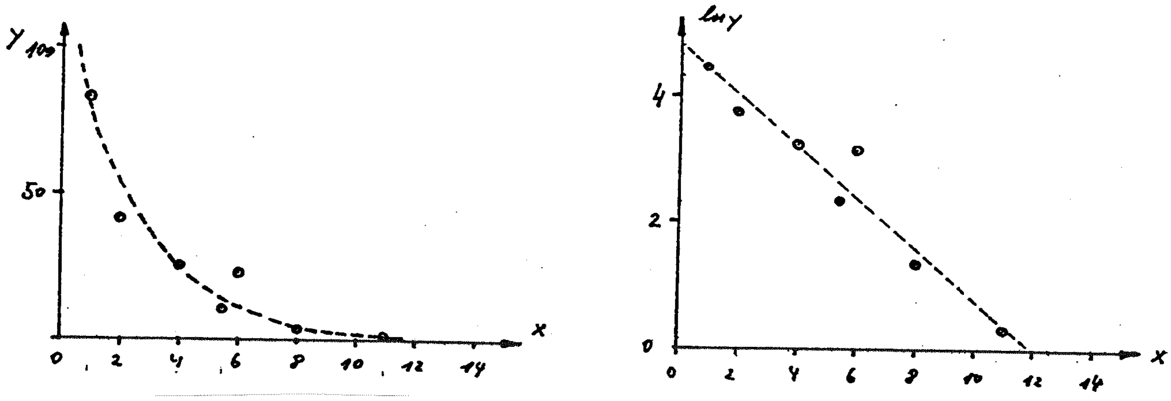


Figure 3.7: Non-linear model function (exponential function) in normal and semi-logarithmic coordinate systems.

notice that the weighting factors g_k are set equal to one!
This system is easy to solve and we obtain:

$$\begin{aligned}
a &= \exp\left[\left(\sum \ln y_k \cdot \sum x_k^2 - \sum x_k \cdot \sum x_k \ln y_k\right)/D\right] \\
\lambda &= -\left(n \cdot \sum x_k \ln y_k - \sum x_k \cdot \sum \ln y_k\right)/D \\
D &= n \cdot \sum x_k^2 - \left(\sum x_k\right)^2
\end{aligned} \tag{3.18}$$

Example: The 7 points shown in Fig.3.7 have coordinates:

x:	1.	2.	4.	5.5	6.	8.	11.
y:	83.2	41.7	25.1	10.5	22.9	3.8	1.4

For these points, using (3.18) one obtains

$$a = 118.90 \quad \text{und} \quad \lambda = 0.398 \quad .$$

The best-fit curve is therefore

$$f(x) = 118.90\, e^{-0.398x}$$

(this is the dashed curve in Fig.3.7), and as a sum of the square errors one obtains

$$\chi^2_{min} = 307.3 \quad .$$

Remarks:

1. This 'minimal error sum' is obviously the minimum of the sum of the square deviations between the $\ln y_k$ and the $\ln f(x_k)$!
2. A big disadvantage of this linearization method is that it is not possible to determine a statistical weighting factor.
3. The exponential model function is a good example of a *physically relevant* model parameter: $a \cdot \exp(-\lambda x)$ could for example be a radioactive decay curve, with a as starting amplitude, and λ as decay constant.

In [7], P. 330 ff, one can find a lot of other examples to linearize simple non-linear model functions.

A final consideration: try to avoid 'hidden correlations' between the fitting parameters in your model functions. For example the model

$$f(x; a, b, c) = a\,\mathrm{e}^{-bx+c}$$

is not a good one, since the parameters a and c cannot be separated numerically:

$$f(x; a, b, c) = \underbrace{a\,\mathrm{e}^{c}}_{=\text{ only 1 parameter}} \cdot \mathrm{e}^{-bx}\,.$$

3.5.3 The Gauss-Newton (GN) method.

In all those cases, in which a linearization of the model function according to section 3.5.2 is not possible or not desired, we can proceed as follows:

The starting point is the <u>completely general</u> model function

$$f(x; a_1, a_2, \ldots, a_q) \tag{3.19}$$

with the parameters $\mathbf{a} \equiv a_1, a_2, \ldots, a_q$.
The minimization of the weighted error sum

$$\chi^2 = \sum_{k=1}^{n} g_k \left[y_k - f(x_k; \mathbf{a})\right]^2$$

follows this *iterative* procedure:

1. Choose a set of (*guessed values*) for the parameters:
$$\mathbf{a} = \mathbf{a}^0 \quad .$$

2. *Expand the model function in a Taylor series* of $\mathbf{a}$ around $\mathbf{a}^0$, and truncate the expansion after the linear term:
$$f(x; \mathbf{a}) \approx f(x; \mathbf{a}^0) + \sum_{l=1}^{q} \left(\frac{\partial f(x; \mathbf{a})}{\partial a_l}\right)_{\mathbf{a}=\mathbf{a}^0} \cdot (a_l - a_l^0) \quad .$$

3. Insert this (approximate) *linearised model function* in the LSQ equations:
$$\chi^2 = \sum_{k=1}^{n} g_k \left[y_k - f(x_k; \mathbf{a}^0) - \sum_{l=1}^{q} \left(\frac{\partial f(x_k; \mathbf{a})}{\partial a_l}\right)_{\mathbf{a}=\mathbf{a}^0} \cdot (a_l - a_l^0)\right]^2 \quad ,$$
We now introduce the notations:
$$df_{k,l} \equiv \left(\frac{\partial f(x_k; \mathbf{a})}{\partial a_l}\right)_{\mathbf{a}=\mathbf{a}^0} \quad \text{and} \quad f_k \equiv f(x_k; \mathbf{a}^0).$$

4. Take the derivative of χ^2 with respect to the model parameters, and set them to zero:
$$\frac{\partial \chi^2}{\partial a_j} = -2 \sum_{k=1}^{n} g_k \left[y_k - f_k - \sum_{l=1}^{q} df_{k,l}(a_l - a_l^0)\right] \cdot df_{k,j} = 0$$
for $j = 1, \ldots, q$. As a result we obtain a linear, inhomogeneous set of equations for the q expressions $(a_l - a_l^0)$:
$$A \cdot (\mathbf{a} - \mathbf{a}^0) = \beta$$
with
$$A = [\alpha_{ij}] \quad \alpha_{ij} = \sum_{k=1}^{n} g_k df_{k,i} df_{k,j} \quad \text{and} \quad \beta_i = \sum_{k=1}^{n} g_k (y_k - f_k) df_{k,i} \quad . \tag{3.20}$$

5. The solutions of this system, $(a_l - a_l^0)$, can now be regarded as differences between the *guessed values* and the *improved values* of the desired model parameters:

$$a_l - a_l^0 \equiv \Delta a_l \rightarrow a_l^1 = a_l^0 + \Delta a_l \quad (l = 1, \ldots, q) \quad .$$

6. Repeat steps $(2 \rightarrow 3 \rightarrow 4 \rightarrow 5)$ with this *improved set of parameters.* In this way (assuming that the procedure is converging) it is possible to *iteratively improve* the model parameters:

$$a_l^t = a_l^{t-1} + \Delta a_l^t \quad (t = 1, 2, \ldots) \tag{3.21}$$

7. As in all iterative methods, also in this case we need to specify an *exit criterion.* Several criteria are discussed in literature. One (not always the best!) is the following:

   The iteration is interrupted, if <u>all</u> parameters satisfy a given condition on the relative precision:

$$\mid a_l^t - a_l^{t-1} \mid < \mid a_l^t \mid \cdot \epsilon \tag{3.22}$$

   (ϵ = precision) or if the number of iterations reaches a fixed maximum value.

8. Once we have attained the desired precision, we can calculate the normal matrix of the problem. The i, j-th coefficients read, according to (3.6):

$$\frac{1}{2} \frac{\partial^2 \chi^2}{\partial a_i \partial a_j} \quad .$$

   With a non-linear model function we obtain:

$$\frac{1}{2} \frac{\partial^2 \chi^2}{\partial a_i \partial a_j} = \sum_{k=1}^{n} g_k \left[\frac{\partial f(x_k; \mathbf{a})}{\partial a_i} \frac{\partial f(x_k; \mathbf{a})}{\partial a_j} - (y_k - f(x_k; \mathbf{a})) \frac{\partial^2 f(x_k; \mathbf{a})}{\partial a_i \partial a_j} \right] \quad .$$

   In most programs which employ the non-linear LSQ approximation the second term in parenthesis in the equation above is neglected. This is justified, because if the iteration is succesful the quantity $[y_k - f(x_k; \mathbf{a})]$ gets smaller and smaller. Using the abbreviations one obtains thus:

$$\frac{1}{2} \frac{\partial^2 \chi^2}{\partial a_i \partial a_j} \approx \sum_{k=1}^{n} g_k df_{k,i} df_{k,j} \quad . \tag{3.23}$$

   In this approximation, like in the linear model, the normal matrix coincides with the matrix of coefficients of the LSQ system!

9. Inversion of the normal matrix = covariance matrix. This matrix contains the statistical information about the fit and the fitting parameters.

3.5.4 Convergence problems in the GN method. Marquardt's variant.

A primary requirement for applying a numerical method in practice is its *stability.* This is particularly important for iterative methods. It is crucial to ensure that the method *converges*, also in cases in which *the starting values for the iteration have not been chosen optimally.*

The following example shows that the GN method does not satisfy this requirement at all.

The starting point for this test is the function

$$f(x) = 10e^{-3x} + 5e^{-x/2} \quad .$$

This corresponds to the parametrised function

$$f(x; a_1, a_2, a_3, a_4) = a_1 e^{-a_3 x} + a_2 e^{-a_4 x} \tag{3.24}$$

with the (exact) parameters

$$a_1 = 10. \quad a_2 = 5. \quad a_3 = 3. \quad a_4 = 0.5 \quad .$$

We can now employ this function to generate a *test data set* for the GN algorithm:

```
 10 Data values:
             X              Y=F(X)
  1     .1000000E+01    .3530524E+01
  2     .2000000E+01    .1864185E+01
  3     .3000000E+01    .1116885E+01
  4     .4000000E+01    .6767378E+00
  5     .5000000E+01    .4104280E+00
  6     .6000000E+01    .2489355E+00
  7     .7000000E+01    .1509869E+00
  8     .8000000E+01    .9157819E-01
  9     .9000000E+01    .5554498E-01
 10     .1000000E+02    .3368973E-01
```

If we regard these data as coordinates of points for a LSQ problem with (3.24) as model function, the GN process should return as a result of the iteration the following values:

$$a_1 \to 10. \quad a_2 \to 5. \quad a_3 \to 3. \quad a_4 \to 0.5$$

In the following we will study the convergence behaviour of a program based on the algorithm introduced in section 3.5.3. For this, we will generate different *guessed values* for the a_3 and a_4, keeping a_1 and a_2 always fixed to their starting values 9. und 4.
The results of this test are collected in Fig.3.8.

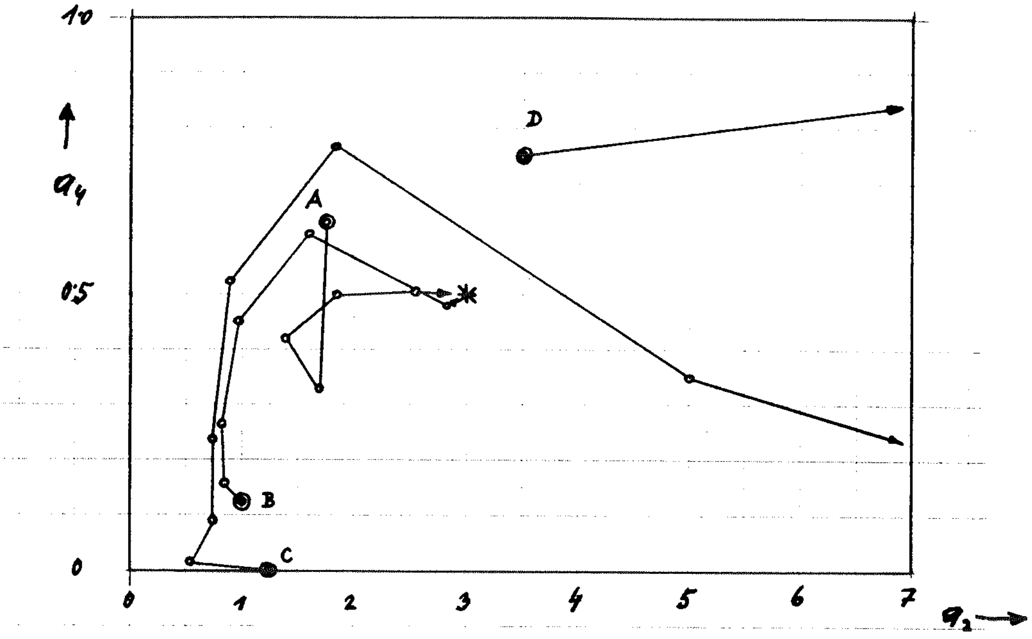


Figure 3.8: Convergence of the Gauss-Newton algorithm.

The results of this test are clearly very unsatisfactory! It is in fact extremely difficult to estimate the convergence domain for the method. We could of course change the initial *guessed values* until the iteration converges, but this is of course not a viable method!

A way out of this dilemma was found in 1963 from D.W. Marquardt [1].

In the following we will present the results of his studies, without entering into the mathematical details of the derivations.

The starting point of the *Marquardt's variant of the GN method* is the linear system (3.20):

$$\begin{pmatrix} \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1q} \\ \alpha_{12} & \alpha_{22} & \cdots & \alpha_{2q} \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ \alpha_{1q} & \alpha_{2q} & \cdots & \alpha_{qq} \end{pmatrix} \cdot \begin{pmatrix} \Delta a_1 \\ \Delta a_2 \\ \cdot \\ \cdot \\ \cdot \\ \Delta a_q \end{pmatrix} = \begin{pmatrix} \beta_1 \\ \beta_2 \\ \cdot \\ \cdot \\ \cdot \\ \beta_q \end{pmatrix}$$

In matrix form this reads:

$$A \cdot \Delta\mathbf{a} = \beta$$

Systems of equations of this type are to be solved with the GN algorithm. *In the Marquardt's variant, instead of the original set of equations we solve the following system*

$$(A + \lambda D) \cdot \Delta\mathbf{a} = \beta \quad , \tag{3.25}$$

where D is a diagonal matrix of the form

$$d_{ii} = \alpha_{ii} \quad (i = 1, \ldots, q). \tag{3.26}$$

[1] D.W. Marquardt, J. Soc. Ind. Appl. Math. **11**,431 (1963)

The quantity λ is still undefined at the moment.

Let us assume that the t-th iteration step of the GN process has given the square error sum χ_t^2. The next iteration step returns the value χ_{t+1}^2. It can happen that:

$$\chi_{t+1}^2 > \chi_t^2 \quad .$$

Under certain circumstances (see the previous test, starting point C) this can lead to a divergence of the iteration process.

The general statement for which Marquardt was able to give a rigorous mathematical proof is the following:
It is always possible to choose the quantity λ in (3.25) in such a way that:

$$\chi_{t+1}^2 \le \chi_t^2$$

This ensures the convergence of the method.

In order to demonstrate the effect of increasing the (positive) quantity λ, we start from a system of equations (3.25), assuming that the model we employ has only 2 parameters:

$$\begin{pmatrix} \alpha_{11}(1+\lambda) & \alpha_{12} \\ \alpha_{12} & \alpha_{22}(1+\lambda) \end{pmatrix} \cdot \begin{pmatrix} \Delta a_1 \\ \Delta a_2 \end{pmatrix} = \begin{pmatrix} \beta_1 \\ \beta_2 \end{pmatrix}$$

The analytical solutions of these systems are:

$$\Delta a_1(\lambda) = \frac{\beta_1 \alpha_{22}(1+\lambda) - \beta_2 \alpha_{12}}{\alpha_{11}\alpha_{22}(1+\lambda)^2 - \alpha_{12}^2} \quad \text{and} \quad \Delta a_2(\lambda) = \frac{\beta_2 \alpha_{11}(1+\lambda) - \beta_1 \alpha_{12}}{\alpha_{11}\alpha_{22}(1+\lambda)^2 - \alpha_{12}^2} \quad .$$

This means:

- For $\lambda \to 0$ we recover the GN method.
- For $\lambda \to \infty$ Δa_1 and Δa_2 go to zero!

Additionally, one realises immediately that the ratio of the correction coefficients $\Delta a_1/\Delta a_2$ is also a function of λ. This means that not only do the corrections get smaller with the variation of λ, but also the direction of the correction changes: we are exploring the vicinity of the origin.

We can thus draw the following conclusions:

- *An increase of the Marquardt's parameter λ leads to a reduction of the correction for the model parameters, and therefore decreases the convergence speed.*
- *On the other hand a (sufficient) increase of λ leads to the monotone decrease of the error sums*:

$$\chi_1^2 \ge \chi_2^2 \ge \chi_3^2 \cdots \ge \chi_t^2 \ge \chi_{t+1}^2 \cdots$$

A possible method to keep λ as small as possible (in order to have a fast convergence) and at the same time large enough to ensure convergence, is the so-called *Marquardt's strategy*:

1. Start the iteration with a small (positive) value for λ (in the following program, for example, $\lambda = 0.0003$).
2. Before each new iteration step, λ is reduced by a fixed – arbitrary – factor to ensure a fast convergence (in the following program, this factor is set to five).
3. In case $\chi^2_{t+1} > \chi^2_t$, λ is increased by a fixed factor (in the following program 5) and the calculation of the set of equations (3.25) is repeated until $\chi^2_{t+1} \leq \chi^2_t$. At this point
4. next iteration

If we test the *GN algorithm with the Marquardt variant* [2] using the data of the test above, we obtain the results in Fig.3.9. As we can see, the method converges *independently of the starting points C and D*, withouth observable divergences (see with Fig.3.8). The following lines show the results of the calculation for the D starting point; the stars * indicate the points where the Marquardt's parameter λ is increased:

```
 Starting point D    NO MARQUARDT VARIANT:
 =========================================
  t       chisq             a1            a2            a3            a4

  0   .368339E+01   .900000E+01   .400000E+01   .350000E+01   .750000E+00
                    Stop of the execution due to exponent overflow!

 Starting point D    WITH MARQUARDT VARIANT:
 ========================================
 t       chisq             a1            a2            a3            a4

 0    .368339E+01      .900000E+01   .400000E+01   .350000E+01   .750000E+00
*
*
*
 1    .327945E+01      .116540E+02   .478102E+01   .326392E+01   .298864E+00
 2    .221123E+00      .139565E+02   .407836E+01   .262388E+01   .386921E+00
 3    .660544E-02      .776914E+01   .456550E+01   .253007E+01   .468859E+00
 4    .383322E-04      .750890E+01   .492223E+01   .264024E+01   .496633E+00
*
 5    .708586E-05      .823696E+01   .497283E+01   .278430E+01   .498760E+00
*
 6    .320911E-05      .885081E+01   .498274E+01   .286453E+01   .499211E+00
 7    .315072E-05      .961475E+01   .499580E+01   .296028E+01   .499807E+00
 8    .101244E-06      .996144E+01   .499965E+01   .299644E+01   .499984E+00
 9    .147690E-10      .999937E+01   .499999E+01   .299994E+01   .500000E+00
10    .903999E-13      .100000E+02   .500000E+01   .300000E+01   .500000E+00
11    .340838E-13      .999999E+01   .500000E+01   .300000E+01   .500000E+00
```

[2] This algorithm is referred to as 'Levenberg-Marquardt Method' in literature

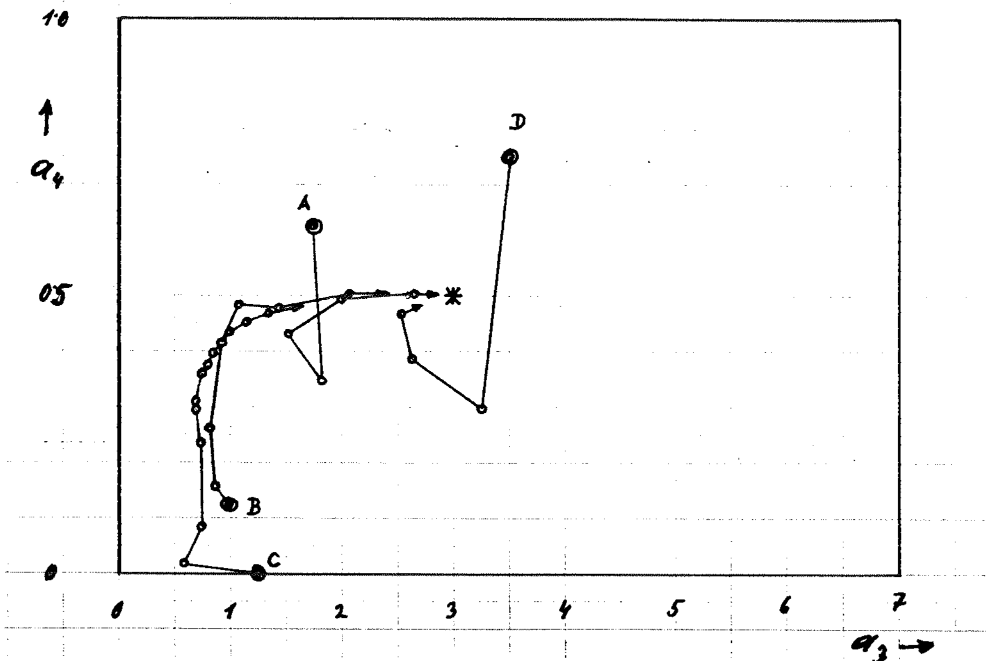


Figure 3.9: Convergence of the Gauss-Newton-Marquardt algorithm.

3.5.5 The program MRQMIN.

Source: [9], P.526f; [10], P. 683ff with modifications.

The program MRQMIN (MarQuardt MINimization) performs an iteration step in the Gauss-Newton-Marquardt method.

INPUT parameters:

X( ),Y( ): Coordinates of the data points.

SIG( ): Standard deviations of the y's.

NDATA: Number of data points.

MA: Number of parameters of the model function.

A( ): Array containing the (*guessed values*) for the model parameters.

ALAMDA: Control parameter, Marquardt's parameter λ.

OUTPUT parameters:

A( ): Array containing the *optimized* fitting parameters.

ALPHA(,): Matrix of coefficients for the linear system (3.20).

COVAR(,): Covariance matrix.

BETA(): Inhomog. vector of the linear system (3.20).

CHISQ: Weighted square error sum χ^2.

OCHISQ: Weighted error sum of the previous iteration step.

ALAMDA: Current value of the Marquardt's parameter.

VMAR: Logical value that shows if the iteration step lead to a decrease in χ^2(VMAR=false) or not (VMAR=true).

Internal Arrays:

NORMAL(,): Marquardt's matrix of coefficients (3.25).

DA(): Correction vector for the parameter.

VECTOR(), SOL() .

Required functions:

MRQCOF: Calculates the matrix ALPHA and of the vector BETA according to (3.19), as well as of the weighted error sum CHISQ.

LUDCMP und LUBKSB: Solves the linear set of equations (3.25), i.e. inverts the normal matrix.

Structure of the program:

1. Begin the iteration with a given ALAMDA. ALAMDA is set to the value 0.0003,[3] and then MRQCOF is called for the first time. The coefficients of the system of equations (3.25) are calculated from the two arrays ALPHA(,) and BETA( ). This system is solved with LU-Decomposition.

2. If ALAMDA is zero (last call of MRQMIN!) the covariance matrix COVAR(,) is calculated inverting the normal matrix NORMAL(,). End of the calculation.

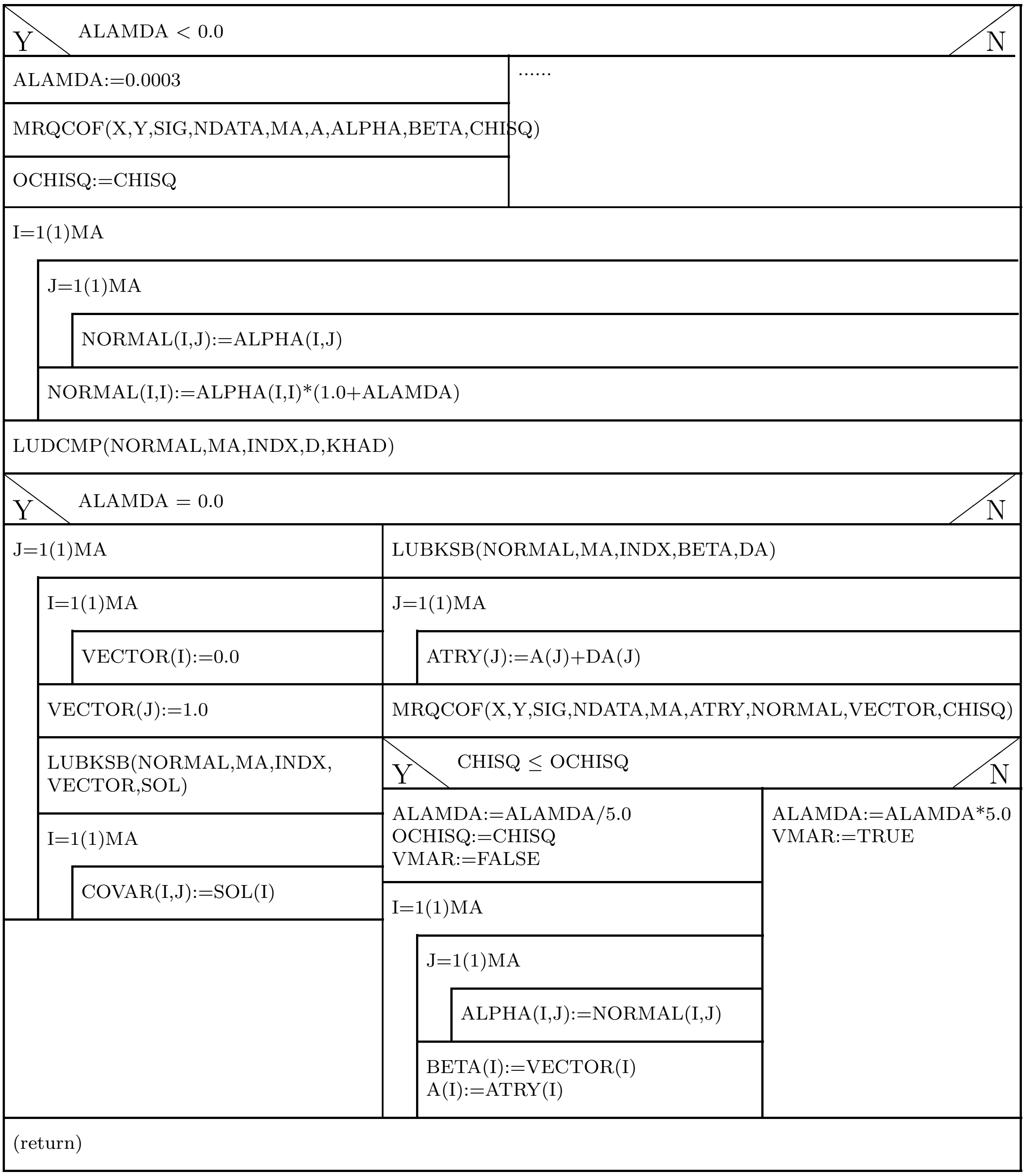
3. If ALAMDA > zero, the improved fitting parameters (ATRY) are calculated. New call of MRQCOF.

4. The new error sum CHISQ given by MRQCOF is calculated and compared with the old error sum OCHISQ:

[3] This starting value for λ, as well as the other factor given below (=5) , is arbitrary; the value we give here generally leads to good results.

- CHISQ ≤ OCHISQ: The iteration step was succesful. The logical variable MVAR is set to 'false', and ATRY( ) is saved in A(). ALAMBA is divided by the factor 5.
- CHISQ > OCHISQ: The iteration step was *not* succesful. The logical variable is set to 'true', and ALAMDA is multiplied by the factor 5.

Structure chart 12 — MRQMIN(X,Y,SIG,NDATA,MA,A,ALPHA,COVAR, BETA,CHISQ,OCHISQ,ALAMDA,VMAR)



```
ALAMDA < 0.0
  Y:
    ALAMDA:=0.0003
    MRQCOF(X,Y,SIG,NDATA,MA,A,ALPHA,BETA,CHISQ)
    OCHISQ:=CHISQ
  N:
    ......
I=1(1)MA
    J=1(1)MA
        NORMAL(I,J):=ALPHA(I,J)
    NORMAL(I,I):=ALPHA(I,I)*(1.0+ALAMDA)
LUDCMP(NORMAL,MA,INDX,D,KHAD)
ALAMDA = 0.0
  Y:
    J=1(1)MA
        I=1(1)MA
            VECTOR(I):=0.0
        VECTOR(J):=1.0
        LUBKSB(NORMAL,MA,INDX,
        VECTOR,SOL)
        I=1(1)MA
            COVAR(I,J):=SOL(I)
  N:
    LUBKSB(NORMAL,MA,INDX,BETA,DA)
    J=1(1)MA
        ATRY(J):=A(J)+DA(J)
    MRQCOF(X,Y,SIG,NDATA,MA,ATRY,NORMAL,VECTOR,CHISQ)
    CHISQ ≤ OCHISQ
      Y:
        ALAMDA:=ALAMDA/5.0
        OCHISQ:=CHISQ
        VMAR:=FALSE
        I=1(1)MA
            J=1(1)MA
                ALPHA(I,J):=NORMAL(I,J)
            BETA(I):=VECTOR(I)
            A(I):=ATRY(I)
      N:
        ALAMDA:=ALAMDA*5.0
        VMAR:=TRUE
(return)
```

3.5.6 The program MRQCOF.

Source: [9], P.527f; [10], P. 687.

The program MRQCOF (MaRQuardt COeFficients) calculates the matrix ALPHA and the vector BETA through (3.20) as well as the weighted square error sum χ^2.

INPUT parameters:

X( ),Y( ),SIG( ),NDATA,MA,A( ): see description of MRQMIN, sect. 3.5.5.

OUTPUT parameters:

ALPHA(,),BETA( ): Matrix ALPHA and Vector BETA according to (3.20).

CHISQ: weighted square error sum χ^2.

Required functions:

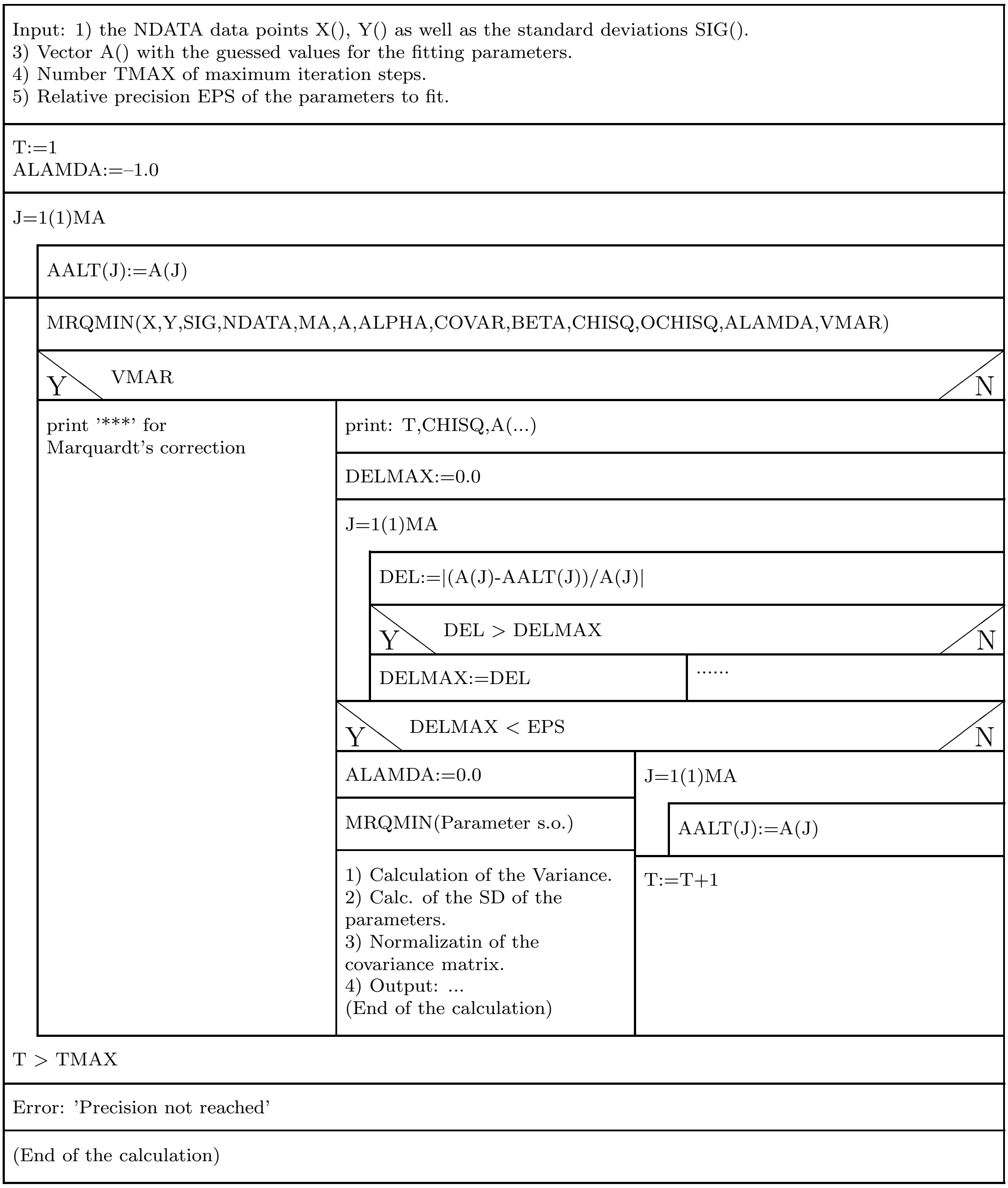
FUNCS: Function that, for given arguments X and model parameters A(), returns the functional value Y for the model function and all ist partial derivatives with respect to the model parameters. These MA derivatives are saved in the array DYDA().

Structure chart 13 — MRQCOF(X,Y,SIG,NDATA,MA,A,ALPHA,BETA,CHISQ)

```
I=1(1)MA
    J=1(1)I
        ALPHA(I,J):=0.0
    BETA(I):=0.0
CHISQ:=0.0
K=1(1)NDATA
    FUNCS(X(K),A,YMOD,DYDA,MA)
    G:=1.0/SIG(K)/SIG(K)
    DY:=Y(K)-YMOD
    I=1(1)MA
        J=1(1)I
            ALPHA(I,J):=ALPHA(I,J) + G*DYDA(I)*DYDA(J)
        BETA(I):=BETA(I) + G*DY*DYDA(I)
    CHISQ:=CHISQ + G*DY*DY
I=2(1)MA
    J=1(1)I-1
        ALPHA(J,I):=ALPHA(I,J)
(return)
```

3.5.7 Use of MRQMIN and MRQCOF

. — .



```
Input: 1) the NDATA data points X(), Y() as well as the standard deviations SIG().
3) Vector A() with the guessed values for the fitting parameters.
4) Number TMAX of maximum iteration steps.
5) Relative precision EPS of the parameters to fit.
T:=1
ALAMDA:=–1.0
J=1(1)MA
    AALT(J):=A(J)
    MRQMIN(X,Y,SIG,NDATA,MA,A,ALPHA,COVAR,BETA,CHISQ,OCHISQ,ALAMDA,VMAR)
    VMAR
    Y:  print '***' for
        Marquardt's correction
    N:  print: T,CHISQ,A(...)
        DELMAX:=0.0
        J=1(1)MA
            DEL:=|(A(J)-AALT(J))/A(J)|
            DEL > DELMAX
            Y:  DELMAX:=DEL
            N:  ......
        DELMAX < EPS
        Y:  ALAMDA:=0.0
            MRQMIN(Parameter s.o.)
            1) Calculation of the Variance.
            2) Calc. of the SD of the
            parameters.
            3) Normalizatin of the
            covariance matrix.
            4) Output: ...
            (End of the calculation)
        N:  J=1(1)MA
                AALT(J):=A(J)
            T:=T+1
T > TMAX
Error: 'Precision not reached'
(End of the calculation)
```

An example.

Let us imagine we have a radioactive substance, which is a mixture of J different radioactive isotopes. All types of nuclei decay with a half-time T_j, starting from an initial activity A_j following the *law of radioactive decay*:

$$A_j = A_j e^{-\ln 2 \cdot t/T_j} \quad .$$

The total activity of the source is therefore

$$A(t) = \sum_{j=1}^{J} A_j e^{-\ln 2 \cdot t/T_j} \quad .$$

The measurement proceeds as follows: we count the number of decays from the source in fixed time interval Δ and save the result. We repeat the counting for a second time interval Δ, and so on. In this way we obtain a sequence of numbers Z_k:

$$Z_k = \int_{t=(k-1)\Delta}^{k\Delta} dt A(t) \quad (k = 1, 2, \ldots) \quad .$$

An experiment of this kind is a typical counting experiment and therefore the measured values are *distributed with a Poisson's law* around their expectation values (see Sect. 4.3.3).

Evaluating the integral above we obtain a model function with

$$Z(k; A_1, \ldots, A_J, T_1, \ldots, T_J) = \sum_{j=1}^{J} \frac{A_j}{\ln 2} T_j \left(e^{+\Delta \ln 2/T_j} - 1 \right) e^{-\Delta \ln 2 \cdot k/T_j} \quad .$$

We can then calculate without problems the derivatives $\partial Z/\partial A_j$ and $\partial Z/\partial T_j$. The corresponding function FUNCS reads:

```
#define DELTA 15.0     // Constant of the experiment
void funcs(double x, double a[], double *z, double dzda[], int ma)
// C-VERSION
{
  int   mterm,j,ind;
  double con,fac1,fac2,fac3,fac4;
  con=DELTA*log(2.0);
  mterm=ma/2;
  *z=0.0;
  for(j=1;j<=mterm;j++) {
    ind=mterm+j;
    fac1=con/a[ind];
    fac2=exp(fac1);
    fac3=fac2-1.0;
    fac4=exp(-fac1*x);

    dzda[j]=a[ind]*fac3*fac4/log(2.0);
    *z=*z + a[j]*dzda[j];
    dzda[ind]=a[j]/log(2.0)*fac4*(fac3*(1.0+fac1*x)-fac1*fac2);
  }
}
```

As we can see, the routine *funcs* is called separately for each value of x. This is not an optimal implementation for MATLAB, since it does not exploit the possibility of this language to operate with vectors in a very efficient way. It is more convenient, as shown in the following example, to declare x from the beginning as a vector, that contains all the values x_i with $i = 1, \ldots, NDATA$:

```
  function [z,dzda] = funcs(x,a,ma,ndata);
% MATLAB VERSION  x is a vector with ndata components

  dzda=zeros(ma,ndata);
  delta=15.0;             % measuring time
  con=delta*log(2);
  mterm=fix(ma/2);

    z=zeros(1,ndata);
    for j=1:mterm
      fac1=con/a(mterm+j);
      fac2=exp(fac1);
      fac3=fac2-1;
      fac4=exp(-fac1*x);

      dzda(j,:)=a(mterm+j)*fac3.*fac4/log(2);
      z=z+a(j)*dzda(j,:);
      dzda(mterm+j,:)=a(j)/log(2).*fac4.*(fac3*(1+fac1*x)-fac1*fac2);
    end
```

A concrete example:

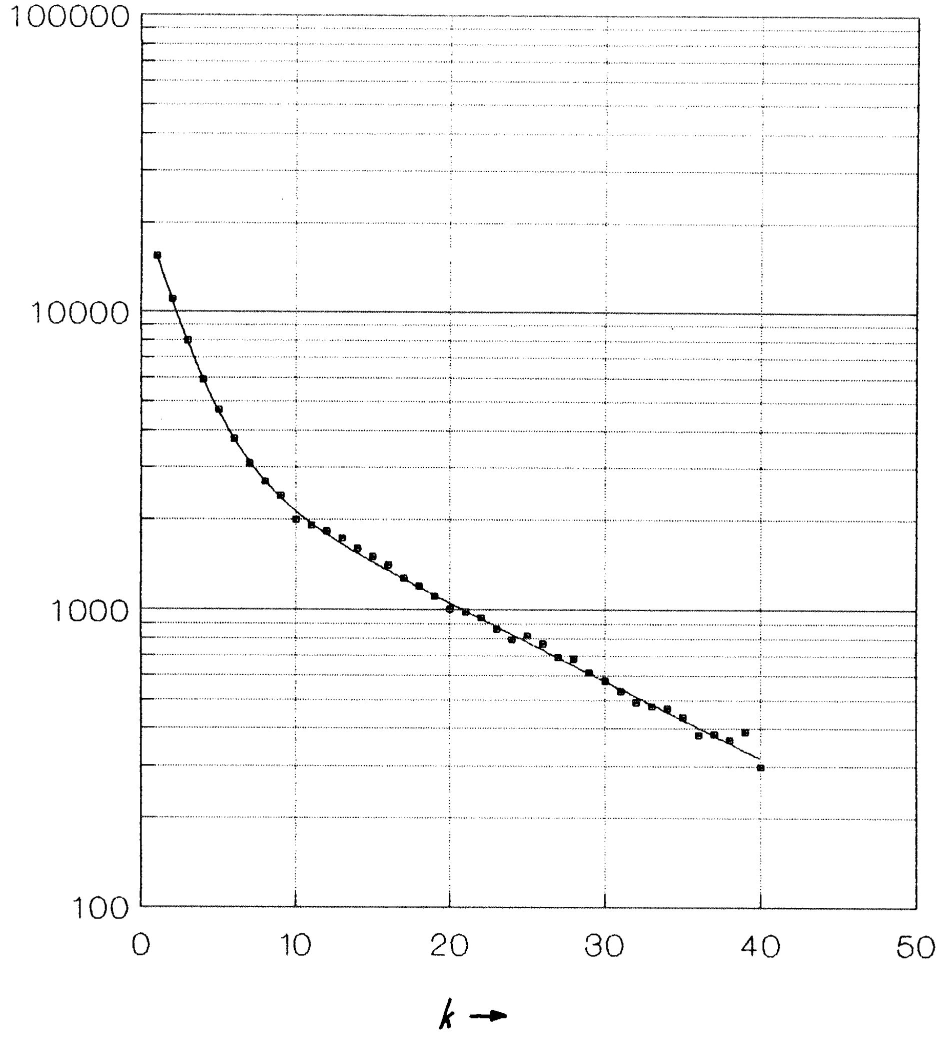
We measure the decay rates of a substance which consists of two components, therefore $J = 2$. We perform 40 measurements, over several intervals of time Δ, of 15 seconds each.

```
  40 Data values:

   1  15376.0            21      981.0
   2  10903.0            22      939.0
   3   7950.0            23      857.0
   4   5865.0            24      790.0
   5   4653.0            25      814.0
   6   3721.0            26      766.0
   7   3089.0            27      691.0
   8   2683.0            28      681.0
   9   2396.0            29      614.0
  10   1992.0            30      576.0
  11   1910.0            31      529.0
  12   1820.0            32      488.0
  13   1726.0            33      472.0
  14   1600.0            34      464.0
  15   1495.0            35      434.0
  16   1410.0            36      380.0
  17   1271.0            37      382.0
  18   1197.0            38      365.0
  19   1106.0            39      387.0
  20   1004.0            40      296.0
```

These are the iterations of the Gauss Newton Marquardt's method:

```
   t     chisq         A1         A2        T1        T2

   0   196876.304   2000.000   500.000   30.000   200.000
   1     1233.069    976.760   237.577   26.378   184.784
   2       45.645    998.414   229.228   22.949   172.341
   3       43.535   1005.360   226.315   23.159   173.250
   4       43.535   1005.458   226.349   23.153   173.245
   5       43.535   1005.457   226.348   23.153   173.246
Convergence is attained!

Variance=  1.209  (should be between 0.764 and 1.236)

          Model parameter        SD

  A1         1005.457          10.182
  A2          226.348           4.129
  T1(s)        23.153           0.353
  T2(s)       173.246           2.320

 The correlation matrix:
           A1        A2        T1        T2

  A1      1.0000   -0.0494   -0.4642    0.0811
  A2     -0.0494    1.0000   -0.7345   -0.9370
  T1     -0.4642   -0.7345    1.0000    0.6405
  T2      0.0811   -0.9370    0.6405    1.0000
```

We conclude with one figure that shows the given data points and the relative fitted model function:


100000
10000
1000
100
0
10
20
30
40
50
$k \rightarrow$

3.6 Add-ons:

Constraints:

In the following we will briefly describe the method for a least square fit, in case the parameters of the model function have to satisfy given linear *constraints.* This is a very common requirement.

Let us imagine we have a model function with q parameters; there can be at most $q - 1$ constraints. If these constraints are linear in the parameters, they can be written as follows:

$$\sum_{l=1}^{q} \gamma_{t,l}\, a_l \overset{!}{=} \delta_t \qquad (t = 1, 2, \ldots, L < q)\,. \tag{3.27}$$

There are thus L constraints, where the $\gamma_{t,l}$ and δ_t are fixed quantities.

In this case the basic equation for the LSQ method takes the following form

$$\chi^2 = \sum_{k=1}^{n} g_k \left[y_k - f(x_k; a_1, \ldots, a_q)\right]^2 + \sum_{t=1}^{L} \mu_t \left[\sum_{l=1}^{q} \gamma_{t,l}\, a_l - \delta_t\right] \quad \longrightarrow \quad \text{Min.}\,, \tag{3.28}$$

we have thus introduced L additional fitting parameters μ_t (called *Lagrange* parameters), which in many cases have no physical meaning.

Least-squares when both variables have uncertainties:

An important problem in the analysis of the experimental data through the LSQ method is that in many cases not only the y, but also the x data, are affected by experimental errors. A typical example is the measurement of some physical quantity as a function of time: in this case not only the quantity which is measured, but also the time (in abscissas) will be affected by statistical errors.

In this kind of situations the use of *standard LSQ methods*, like those illustrated in the previous sections of this chapter, is not possible. We have to use, in this case, the so-called *effective variance method.* On this subject, we refer the reader to the following reference:

J. Orear, Am. J. Phys. **50**, 912 (1982)

Chapter 4

Numerical solution of transcendental equations.

4.1 The basic problem.

Let us consider a real-valued function $F(x)$. We seek the real values of x for which:

$$F(x) = 0. \tag{4.1}$$

The values of x that obey this relation are called *solutions, zeroes, roots* of (4.1).

There are many methods in literature to treat this problem. In this lecture we will discuss the following:

- Iterative methods (Newton-Raphson method; Regula Falsi).
- The bissection method.
- The numerical treatment of non-linear systems of equations.

Note: The fact that in the title of this chapter we put the emphasis on *transcendental* equations obviously does NOT mean that we wish to exclude *algebraic* equations of the type:

$$F(x) \equiv P_m(x) = \sum_{j=1}^{m} \alpha_j x^{j-1} = 0$$

Algebraic equations are considered a special cases of transcendental equations.

In literature, one can find many specialized methods to determine numerically the zeroes of algebraic equations (polynomials) – for example, the method of Lobatschewski and Graeffe [20], p.60ff etc.). We will not treat them in this lecture.

4.2 Iterative methods.

4.2.1 General concepts.

The equation $F(x) = 0$ can always (or almost always) be reduced into the form:

$$x = f(x). \tag{4.2}$$

In this way one obtains a typical *iteration ansatz* for finding the zeroes:

$$\begin{array}{rcl} x_0 & & \text{Starting value} \\ x_1 & = & f(x_0) \\ x_2 & = & f(x_1) \\ & . & \\ & . & \\ x_{t+1} & = & f(x_t) \qquad (t = 0, 1, 2, \ldots) \end{array} \tag{4.3}$$

If the method is converging this iteration can lead arbitrarily close to the exact solution of the problem (within roundoff error). We have in fact:

$$\lim_{t \to \infty} x_t \to x_{exact}$$

There is no warranty that this method will converge. On the other hand, the convergence *depends strongly from the way in which (4.1) is reformulated into (4.2)*, as we will show in the examples that follow.

Let us imagine that we are looking for the zeroes of the function $F(x) = x^3 - x - 5$ *in the vicinity of x=2*. We will now show that if the reformulation of $F(x) = 0$ into (4.2) is performed in different ways, the convergence behaviour is very different:

$$a) \quad x = x^3 - 5 \qquad b) \quad x = \frac{5}{x^2 - 1} \qquad c) \quad x = \sqrt[3]{x + 5}$$

```
                x0 = 2.0

t          (a)            (b)            (c)
-----------------------------------------------
0            2          2              2
1            3            1.6667       1.9129
2           22            2.8125       1.9050
3        10643            0.7236       1.9042
4            .          -10.4944       1.9042
.            .            .              .
.            .            .              .
           DIV          DIV            CONV
```

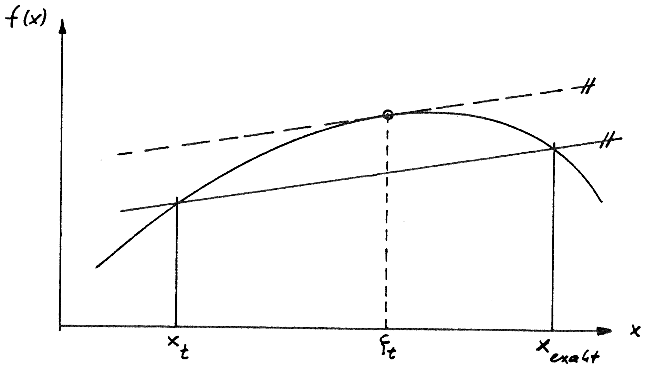


Figure 4.1: Estimating errors in iterations.

4.2.2 Convergence criteria and estimate of the error.

In order to obtain the relation that ensures the convergence of the iteration we start from Eq. (4.2) for the exact solution

$$x_{exact} = f(x_{exact}). \tag{4.4}$$

At the $(t+1)$-th iteration step we obtain:

$$x_{t+1} = f(x_t) - \delta_t \quad , \tag{4.5}$$

where δ_t represents the roundoff error in the numerical evaluation of the function f for $x = x_t$. Subtracting (4.5) from (4.6 gives

$$x_{exact} - x_{t+1} = f(x_{exact}) - f(x_t) + \delta_t\,.$$

Using the mean-value theorem (see Fig.4.1) we can write:

$$\frac{f(x_{exact}) - f(x_t)}{x_{exact} - x_t} = f'(\xi_t) \quad \text{with} \quad \xi_t \in [x_t, x_{exact}] \quad .$$

From this it follows

$$x_{exact} - x_{t+1} = f'(\xi_t) \cdot (x_{exact} - x_t) + \delta_t \tag{4.6}$$

This equation can step by step reduced to

$$\begin{aligned} x_{exact} - x_{t+1} \quad &= \quad f'(\xi_t)f'(\xi_{t-1}) \cdots f'(\xi_0) \cdot (x_{exact} - x_0) + \\ & \quad +\delta_t + f'(\xi_t)\delta_{t-1} + f'(\xi_t)f'(\xi_{t-1})\delta_{t-2} + \\ & \quad + \cdots + f'(\xi_t)f'(\xi_{t-1}) \cdots f'(\xi_1) \cdot \delta_0 \end{aligned}$$

In order to *estimate the error* we will now assume that:

1. the maximum value of the first derivative of the function in the interval I is m, from which it follows that for $x_0, x_1, \ldots, x_{t+1}, \ldots, x_{exact} \in I$:

$$m = \max_I \mid f'(x) \mid \tag{4.7}$$

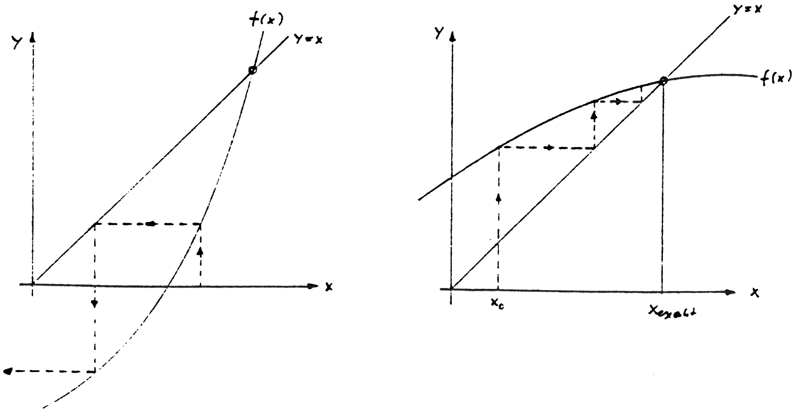


Figure 4.2: Convergence behaviour of the iteration (5.3).

2. *δ does not depend on the index of the iteration*, and therefore for all $t = 0, 1, \dots$ we have:
$$\delta_t \approx \delta \quad .$$

From this we have:

$$\mid x_{exact} - x_{t+1} \mid \leq m^{t+1} \mid x_{exact} - x_0 \mid + \mid \delta \mid (1 + m + m^2 + \cdots m^t)$$

This expression tends to:

$$\lim_{t\to\infty} \mid x_{exact} - x_{t+1} \mid \leq \underbrace{\mid x_{exact} - x_0 \mid \cdot \lim_{t\to\infty} m^{t+1}}_{\text{Methodological error}} + \underbrace{\frac{\mid \delta \mid}{1-m}}_{\text{Roundoff error}} \quad .$$

Both terms diverge for $m \geq 1$, i.e. the *convergence criterion* for the iteration is:
$$0 \leq m < 1 \quad . \tag{4.8}$$

If this relation is satisfied, we have

$$\lim_{t\to\infty} \mid x_{exact} - x_{t+1} \mid \leq \frac{\mid \delta \mid}{1-m} \quad . \tag{4.9}$$

This result is clearly very important! It demonstrates that the iteration method is *numerically stable*: the size of the roundoff errors converges to a limiting value. These relations are illustrated in Figs. 4.2 and 4.3.

In order to *estimate* the error, we can start again from equation (4.6):

$$\begin{aligned}
x_{exact} - x_{t+1} &= f'(\xi_t) \cdot (x_{exact} - x_t) + \delta_t \\
&= f'(\xi_t) \cdot (x_{exact} - x_t + x_{t+1} - x_{t+1}) + \delta_t \\
(x_{exact} - x_{t+1}) \cdot (1 - f'(\xi_t)) &= f'(\xi_t) \cdot (x_{t+1} - x_t) + \delta_t \\
(x_{exact} - x_{t+1}) &= \frac{f'(\xi_t)}{1 - f'(\xi_t)}(x_{t+1} - x_t) + \frac{\delta_t}{1 - f'(\xi_t)} \quad .
\end{aligned}$$

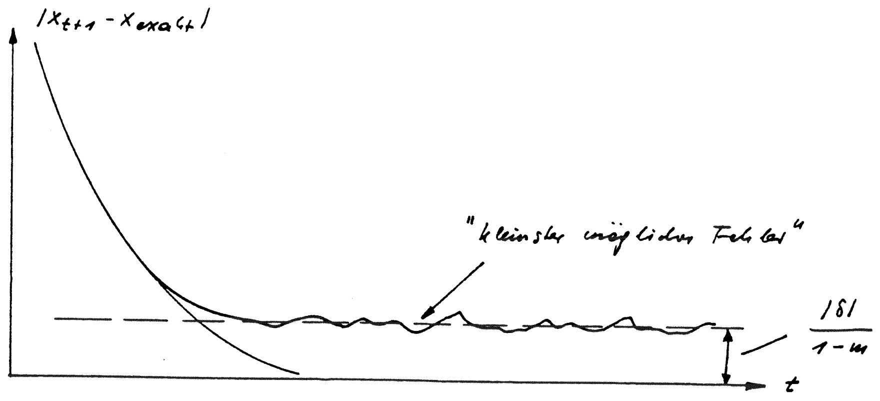


Figure 4.3: Error evolution of the iteration (5.3).

If the iteration converges we obtain:

$$| x_{exact} - x_{t+1} | \leq \frac{m}{1-m} | x_{t+1} - x_t | + \frac{| \delta |}{1-m} \quad . \tag{4.10}$$

If we are able to calculate the maximum value of the first derivative of the function $f(x)$ in the interval I and to estimate the value of δ accordingly, Equation (4.10) can serve as a basis for an error control of the iterative method: we can iterate until the expression on the right hand side of (4.10) becomes smaller than a desired precision ϵ. We only have to remember not to set ϵ to a value smaller than the *limiting precision* given by $| \delta | /(1-m)$!

In practice the determination of m and δ is most of the times not possible or too complicated. In this case we can employ the much easier criterion:

$$| x_{exact} - x_{t+1} | \leq | x_{t+1} - x_t | \quad , \tag{4.11}$$

from which we derive the following

Exit criteria — .

<table>
<tr><td colspan="2">Y |x(t+1)-x(t)| &lt; EPS N</td></tr>
<tr><td>End of the iteration</td><td>Next iteration</td></tr>
</table>

for the iteration.
We must however keep in mind that (4.11) holds only for $m \leq 1/2$, and can be grossly wrong for $1/2 < m < 1$! Otherwise, (4.11) is useful only as long as *the methodological error is larger than the roundoff error!*

The following example should illustrate some of the relations derived before:

We want to determine numerically the zeroes of the following function:

$$F(x) = a + (1-a)x^2 - x \quad .$$

As can be easily verified, this quadratic function has two zeroes:

$$x_1 = 1 \qquad \text{and} \qquad x_2 = \frac{a}{1-a} \quad .$$

We now want to find an approximation for the zero x_1 through the iteration

$$x_{t+1} = a + (1-a)x_t^2.$$

The first derivative of $f(x)$ is $2(1-a)$ in $x = 1$. This is also the maximum value of the derivative in the iteration interval, i.e. we have

$$m = |2(1-a)|\,.$$

On the basis of (4.8) and (4.10) we expect that

- the iteration diverges for $m \geq 1$, i.e. for $a \leq 0.5$ or $a \geq 1.5$;
- the simplified error estimate (4.11) breaks down for $1/2 < m < 1$, i.e. for $a < 0.75$.

<u>Results of this test calculation:</u>

```
Parameter  a = 2.500  > 1.5  i.e. Divergence expected:

   t        x_{t}      x_{ex}-x_{t+1}    x_{t+1}-x_{t}

   0    .1200000E+01
   1    .3400000E+00    .6600000E+00     -.8600000E+00
   2    .2326600E+01   -.1326600E+01      .1986600E+01
   3   -.5619601E+01    .6619601E+01     -.7946201E+01
   4   -.4486988E+02    .4586988E+02     -.3925028E+02
   5   -.3017459E+04    .3018459E+04     -.2972589E+04
   6   -.1365759E+08    .1365759E+08     -.1365457E+08
   7   -.2797945E+15    .2797945E+15     -.2797945E+15
   8   -.1174274E+30    .1174274E+30     -.1174274E+30
   9   -.2068380E+59    .2068380E+59     -.2068380E+59
  10   -.6417295+117    .6417295+117     -.6417295+117
```

```
Parameter  a = 0.600  ===> m=0.8  i.e.: convergence, but
                                    breakdown of (5.11):

   t        x_{t}       x_{ex}-x_{t+1}    x_{t+1}-x_{t}

   0    .6000000E+00
   1    .7440000E+00    .2560000E+00      .1440000E+00
   2    .8214144E+00    .1785856E+00      .7741440E-01
   3    .8698886E+00    .1301114E+00      .4847425E-01
   4    .9026825E+00    .9731750E-01      .3279386E-01
   5    .9259343E+00    .7406572E-01      .2325178E-01
   6    .9429417E+00    .5705828E-01      .1700744E-01
   7    .9556556E+00    .4434437E-01      .1271392E-01
   8    .9653111E+00    .3468892E-01      .9655443E-02
   9    .9727302E+00    .2726981E-01      .7419114E-02
  10    .9784816E+00    .2151839E-01      .5751419E-02


Parameter  a = 1.200  ===> m=0.4  d.h.: convergence and
                                    validity of (5.11):


   0    .6000000E+00
   1    .1128000E+01   -.1280000E+00      .5280000E+00
   2    .9455232E+00    .5447680E-01     -.1824768E+00
   3    .1021197E+01   -.2119718E-01      .7567398E-01
   4    .9914313E+00    .8568734E-02     -.2976591E-01
   5    .1003413E+01   -.3412809E-02      .1198154E-01
   6    .9986325E+00    .1367453E-02     -.4780262E-02
   7    .1000547E+01   -.5466072E-03      .1914060E-02
   8    .9997813E+00    .2187027E-03     -.7653099E-03
   9    .1000087E+01   -.8747150E-04      .3061742E-03
  10    .9999650E+00    .3499013E-04     -.1224616E-03
```

4.3 The Newton Raphson method.

In section 4.2.1 we have mentioned that there are several methods to recast the equation $F(x) = 0$ into the equivalent form $f(x) = x$. A very important way to reformulate the problem is

$$f(x) = x - \frac{F(x)}{F'(x)} \quad , \tag{4.12}$$

Using (4.4) with the prescription:

$$x_{t+1} = x_t - \frac{F(x_t)}{F'(x_t)}. \tag{4.13}$$

we obtain the *Newton Raphson iteration*. The graphical interpretation of this formula ('tangent method') is sufficiently well-known. (see Fig.4.4).

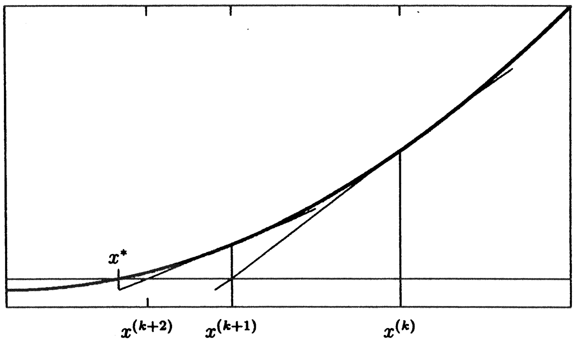


Figure 4.4: Graphical interpretation of the Newton-Raphson iteration.

For the error estimate we can again use (4.10). Here, due to (4.9) and 4.12), m is:

$$m = \max_I \frac{d}{dx}\left(x - \frac{F(x)}{F'(x)}\right) = \max_{[x_0, x_{exact}]}\left(\frac{F(x)F''(x)}{[F'(x)]^2}\right).$$

According to the convergence criterion (4.8) that m must be smaller than one we can also state:

It is difficult to apply the Newton-Raphson iteration if in the vicinity of the zero the slope of $F(x)$ is small. This can happen, for example, if two zeroes lie close to each other.

However, when the Newton-Raphson method works, its convergence is very fast, as shown by the following example.

If we expand the function $F(x)$ around x_t in a Taylor series up to second order, we have:

$$F(x) = F(x_t) + F'(x_t)(x - x_t) + \frac{1}{2}F''(x_t)(x - x_t)^2 .$$

If x is the exact solution we obtain, dividing by $F'(x_t)$:

$$0 = -\frac{F(x_t)}{F'(x_t)} - (x_{exact} - x_t) - \frac{1}{2}\frac{F''(x_t)}{F'(x_t)}(x_{exact} - x_t)^2 .$$

The first term on the right hand side represents the correction term in the Newton-Raphson formula, i.e. we have:

$$0 = x_{t+1} - x_t - x_{exact} + x_t - \frac{1}{2}\frac{F''(x_t)}{F'(x_t)}(x_{exact} - x_t)^2$$

and

$$(x_{exact} - x_{t+1}) = -\frac{1}{2}\frac{F''(x_t)}{F'(x_t)}(x_{exact} - x_t)^2 . \tag{4.14}$$

From this result it follows that the absolute error in the solution at the $(t+1)$-th iteration is proportional to the *square* of the absolute error in the $t-$th

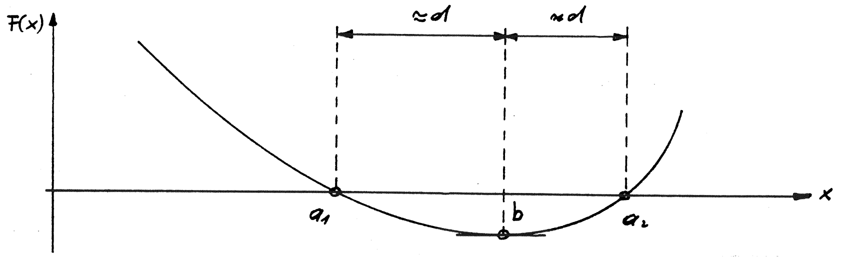


Figure 4.5: Macon's method.

iteration step.
We can thus say: *the Newton-Raphson method converges quadratically.*

A possibility, when pairs of zeroes sit very close to each other, is to employ the

4.3.1 Macon's method.

In general localizing the minimum or maximum of the function between the two zeroes a_1 and a_2 is no problem. If we assume that the value of the abscissas of the extremal value b is known with enough precision, we can expand $F(x)$ in Taylor series around b:

$$F(x) = F(b) + (x-b)F'(b) + \frac{1}{2}(x-b)^2 F''(b) + \cdots \quad . \tag{4.15}$$

We can further assume that, to a first approximation, the two zeroes a_1 and a_2 are located *symmetrically* around b and have therefore a distance d from the extremum. This means:

$$F(b+d) \approx 0 \qquad \text{und} \qquad F(b-d) \approx 0 \quad .$$

Inserted into (4.15) this gives

$$0 \approx F(b) + \frac{1}{2}(+d)^2 F''(b) \qquad 0 \approx F(b) + \frac{1}{2}(-d)^2 F''(b) \quad .$$

These two equations are equivalent and permit to estimate d as:

$$d = \pm\sqrt{-\frac{2F(b)}{F''(b)}} \quad . \tag{4.16}$$

If we now start the Newton-Raphson iteration using $b-d$ or $b+d$ as starting values, we can most of the times avoid convergence problems.

4.3.2 The program RTNEWT.

Source: [9],P. 257 f. with changes.

The subroutine RTNEWT (RooT NEWTon) finds a real zero within a given interval through a Newton-Raphson iteration.

INPUT parameters:

X1,X2: Beginning and end of the interval where the zero lies.

JMAX: Maximum number of iterations.

XACC: Relative precision limit according to Eq.(4.11).

OUTPUT parameters:

RTNEWT: Approximate value of the solution.

ERROR: Error diagnostics:

ERROR = 0: Newton iteration OK.
ERROR = 1: No sign change in [X1,X2]
ERROR = 2: No convergence within JMAX iterations.
ERROR = 3: 'Jumped out of brackets' during Newton.

Internal Variables:

FUNC,DF: $F(x)$ and $F'(x)$. The function that calculates these quantities must be called **FUNC**.

DX: Iteration correction.

Remarks:

- At the beginning, the program tries to understand whether (at least) one of the two zeroes lies in the interval specified [X1,X2]. This is done checking whether the function $F(x)$ changes sign at least once.
- The formula for the *relative* error used in the program leads to convergence problems if the zero of the function is found exactly in $x = 0.0$.
- The first 'emergency exit' of the program RTNEWT ('jumped out of brackets') occurs if during the iterations the program jumps out of the interval specified by the external program that calls the routine (see Fig.4.6, left).
- The second 'emergency exit' ('exceeding maximum iterations') occurs in case of a divergence or also of a too slow convergence, but also in those special cases, in which the iteration gets caught in a loop (see Fig.4.6, right).

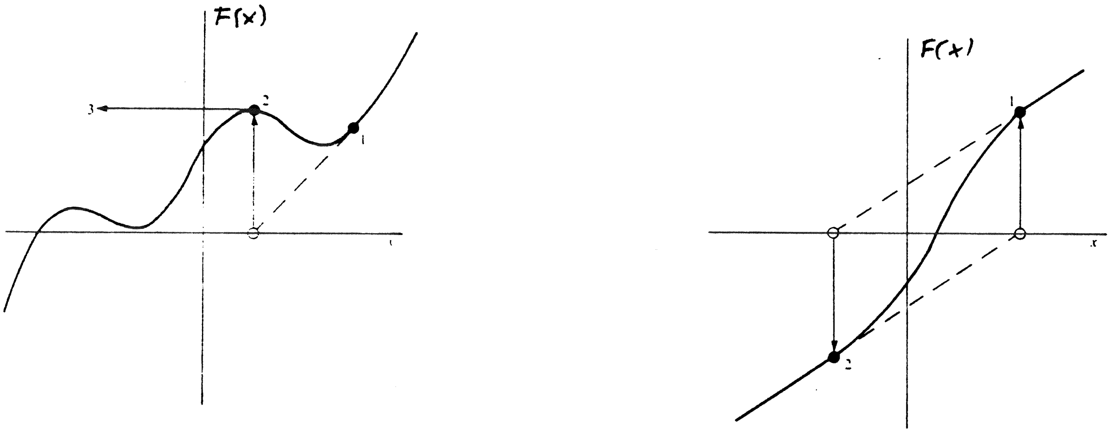


Figure 4.6: Problems in the Newton-Raphson-Iteration.

Structure chart 14 — FUNCTION RTNEWT(X1,X2,JMAX, XACC,ERROR)

```
X:=0.5*(X1+X2)
(FUNC(X1,DF)*FUNC(X2,DF)) > 0.0
  Y: ERROR:=1
     RTNEWT:=X
     print:ERR(1) 'no zero in interval'
     (return)
  N: ......
ERROR:=2
J:=0
    DX:=FUNC(X,DF)/DF
    X:=X-DX
    (X1-X)*(X-X2) < 0.0
      Y: ERROR:=3
      N: |DX/X| < XACC
           Y: ERROR:=0
           N: ......
    J:=J+1
J>JMAX .or. ERROR≠2
ERROR=2
  Y: print: 'ERR(2): no convergence'
  N: ......
ERROR=3
  Y: print: 'ERR(3): jumped out of brackets'
  N: ......
RTNEWT:=X
(return)
```

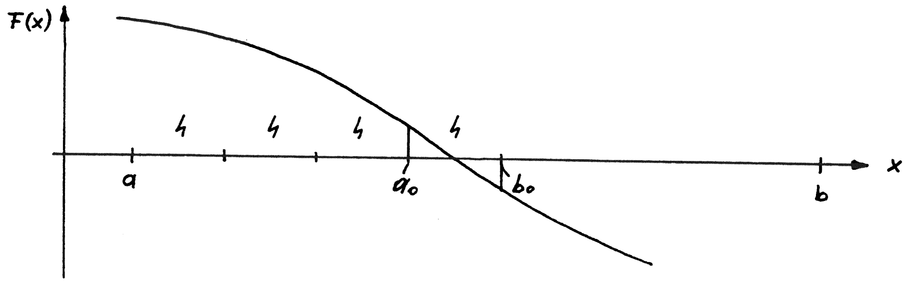


Figure 4.7: Gross search for zeroes.

4.3.3 A test program for RTNEWT.

In general, a function $F(x)$ has more than one real zero. If we want to find them,it is meaningful to combine the Newton-Raphson iteration with a so-called 'gross search'.
The principle of this simple strategy is shown in Fig. 4.7: we perform a scan of the x axis, using intervals of a fixed width h; in this scan, we look for a sign change in $F(x)$. If we find an interval with the property $F(a_0) \cdot F(b_0)$, this means that the interval $[a_0, b_0]$ contains at least one zero. The Newton iteration can begin.
This 'gross search' is also a part of the *bissection method* described in section 4.5.

We now show an example (in C):

```
#include <stdio.h>
#include <math.h>

double func(double x,double *df)
// This function calculates the function values as well as
// the values of the first derivative for the test example 5.3.3.
{
  *df=4*pow(x,3)-27*pow(x,2)-4*x+120;
  return pow(x,4)-9*pow(x,3)-2*pow(x,2)+120*x-130;
}

double rtnewt(double x1, double x2, int jmax, double xacc, int *error)
// This function calculates the real zeroes of the function 'fund'
// in the interval [x1,x2] with the relative precision 'xacc'.
// At the end of the calculation the variable 'error' can take
// one of the following values:


//          error=0       The Newton-Raphson iteration is ok.
//          error=1       No sign change of the function 'funct' in the
//                              intervall [x1,x2].
//          error=2       No convergence within 'jmax' iterations.
//          error=3       During the iteration the value x jumps out
//                         of the interval [x1,x2].
```

```
{
  int j,pause;
  double x,df,dx;

  x=0.5*(x1+x2);

  if((func(x1,&df)*func(x2,&df)) > 0.0) {
    *error=1;
    printf("ERROR(1): no sign change in [x1,x2]\n");
    return x;
  }

  *error=2;
  j=0;

  do {
    dx=func(x,&df)/df;
    x=x-dx;

    if(((x1-x)*(x-x2))<0.0) *error=3;
    else {
      if(fabs(dx/x) < xacc) *error=0;
    }
    j++;
  } while ((j<=jmax) && (*error==2));

  if(*error==2)printf("ERROR(2):  no convergence\n")
  if(*error==3)printf("ERROR(3):  x jumped out of [x1,x2]\n");

  return x;
}


//             ****** main program ******
void main()
{
  int jmax,error,pause;
  double a,b,hgrob,eps,xl,xr,yl,yr,df,zero;

// Parameters for the gross search:
  a=-10.0;  b=10.0;  hgrob=0.5;

// Parameters for the Newton-Raphson iteration:
  jmax=20;
  eps=0.0000001;
  printf("TEST:  NEWTON-RAPHSON-ITERATION WITH GROSS SEARCH\n");
  printf("\n   Gross search interval:       %7.3f  to  %7.3f\n",a,b);
  printf("   Interval width for the gross search: %7.3f\n",hgrob);
  printf("       Par. for Newton:  rel. precision   = %12.10f\n", eps);
  printf("                          max. Iter.step = %4i\n\n", jmax);

// Gross search:
  xl=a;
  yl=func(xl,&df);
  xr=a+hgrob;
  yr=func(xr,&df);

  do {
    if(yl*yr < 0.0) {
// Gross search found a sign change; Newton iteration starts:
      zero=rtnewt(xl,xr,jmax,eps,&error);

      if(error==0)
        printf("  Zero = %9.6f\n",zero);
      else printf("   error(%1i) in RTNEWT\n",error);
    }
    xl=xr;
    yl=yr;
    xr=xl+hgrob;
    yr=func(xr,&df);
  } while(xl <= b);
  printf("\n End of the calculation\n");
}
```

```
*********************************************
TEST:  NEWTON-RAPHSON ITERATION WITH GROSS SEARCH
*********************************************

   Gross search interval:       -10.000  to   10.000
   Width of the interval for the gross search:   0.500
       Par. for Newton:  rel. precision   = 0.0000001000
                          max. Iter.steps =   20

  Zero = -3.600135
  Zero =  1.228589
  Zero =  3.972068
  Zero =  7.399477

End of the calculation.
```

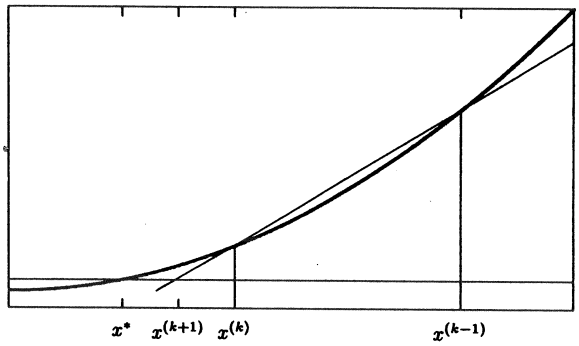


Figure 4.8: Graphical interpretation of the Regula Falsi.

4.4 The Regula Falsi.

This method, which is closely related to the Newton-Raphson method, is often used when the calculation of the derivative of the function $F(x)$ is for some reason complicated. In this case the *differential ratio* in (4.13) is replaced by the *difference ratio*:

$$x_{t+1} = x_t - F(x_t) \cdot \frac{x_t - x_{t-1}}{F(x_t) - F(x_{t-1})} \tag{4.17}$$

Graphically this means that the Newton's *tangent method* is replaced by a *secant method* (see Fig.4.8).

4.5 The bissection method.

In order to identify the real zeroes of the function $F(x)$ in a given interval $[a, b]$, one carries out a *gross localisation* of the zeroes with a given interval width h, starting from a.

Let us assume that the first zero lies in the interval $[a_0, b_0]$ (see Fig.4.7). This means that inside this interval the function $F(x)$ changes sing, i.e. we have:

$$F(a_0) \cdot F(b_0) < 0\,.$$

Once an interval with this property is identified, the actual intervalbissection takes place. First of all, the interval $[a_0, b_0]$ is divided into half:

$$x_0 = \frac{a_0 + b_0}{2}\,.$$

There are now three possibilities:

1. $F(x_0) = 0$: x_0 is (exactly) the desired zero.
(This case will occur rarely due to the roundoff error in the evaluation of the function).

2. $F(a_0) \cdot F(x_0) < 0$ In this case the desired zero lies in the left half interval $[a_0, x_0]$.

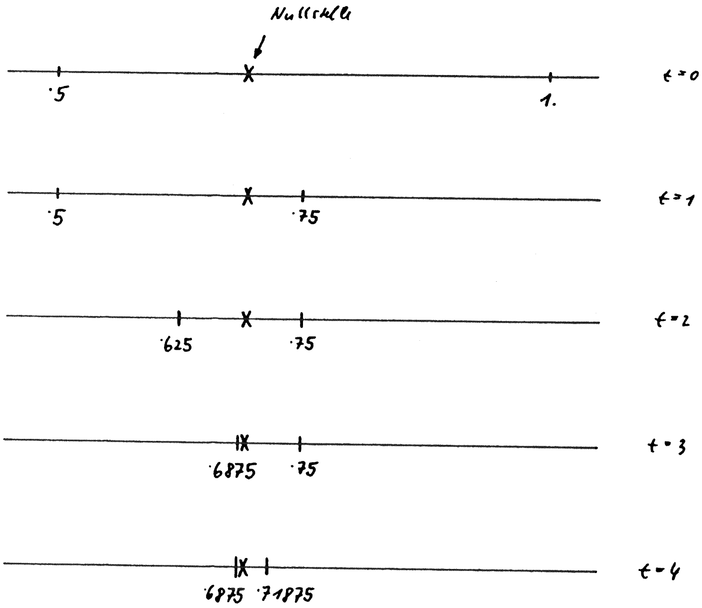
3. $F(a_0) \cdot F(x_0) > 0$ In this case the desired zero lies in the right half interval $[x_0, b_0]$.

In case (1) the zero is found; in the cases (2) and (3) the left or the right half interval are further divided into half. This process is further iterated, until the zero is determined with the required accuracy, i.e. until the current interval width after t divisions ($[a_t, b_t]$) is smaller than the precision with which we wish to determine the zeroes.

We will now illustrate this simple algorithm with an example. We search the zero of the function

$$F(x) = \mathrm{e}^{-x} - \frac{1}{2},$$

which lies exactly in $x_{exact} = \ln 2 = 0.6931472\ldots$ The interval identified through a previous gross search is $[a_0, b_0] = [0.5,1.0]$. The interval bissection proceeds as follows:



It is also important to stress that the method of bissection belongs to a class of numerical methods, for which it is possible to estimate *a priori* the error – at least the absolute one:

$$|x_t - x_{exact}| \leq \frac{1}{2}(b_t - a_t) = \frac{b_0 - a_0}{2^{t+1}}, \tag{4.18}$$

where t is the number of times the interval is divided into half. This formula signifies a disadvantage of this method, namely its slow convergence: in fact, in order to increase absolute precision of the result by 10 %, we need to perform 3.3 calculation steps (see section 4.5.2).

4.5.1 Problems with the bissection method.

In principle this method is very safe. If a zero is located in the interval $[a_0, b_0]$, it can be calculated with the desired accuracy (apart from roundoff errors). There are however some small problems connected with the convergence of this method.

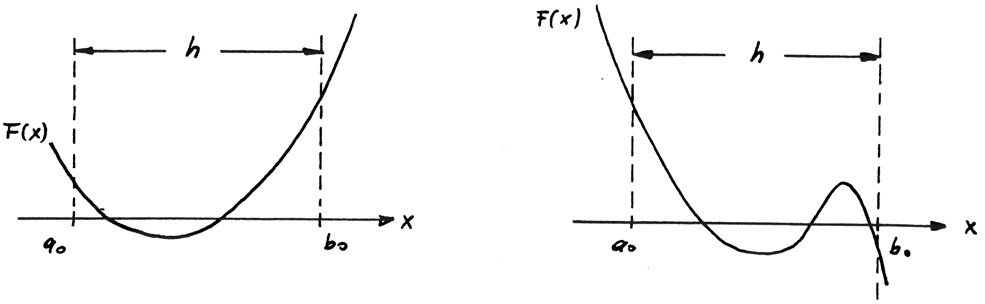


Figure 4.9: Problems connected with the bissection method.

The only source of uncertainity lies in the *choice of the interval width for the gross search.* The criterion: sign change = zero is not always void of problems (see Fig. 4.9):

- In the example shown in the left panel of Fig.4.9, there is no sign change between the two extrema of the interval, although there are zeroes in-between. The same happens every time that the interval $[x_1, x_2]$ contains an *even* number of zeroes.
- In the example shown in Fig.4.9, right *the sign change results from an odd number of zeroes.*

In both cases we have chosen a too large interval width h! This example shows that h acts as a *resolution limit* for this method:

In general we can determine only those zeroes, which are separated by a distance which is larger than the initial width h.

4.5.2 The program INTSCH.

Source: [2],P.281f with changes and simplifications.

INPUT parameters:

INIT,AEND: Beginning and end of the interval for the gross search.

H: Stepsize for the gross search.

PREC: Limit on the error (usually relative error; see remark).

NMAX: Maximum number of zeroes saved in the output array ZERO.

OUTPUT parameters:

ZERO( ): Array that contains the calculated real zeroes.

N: Number of zeroes calculated by INTSCH.

Error diagnostics:

N=0:   no zeroes found.

N=NMAX+1:   more than NMAX zeroes found, but only the first NMAX zeroes are saved in the array ZERO.

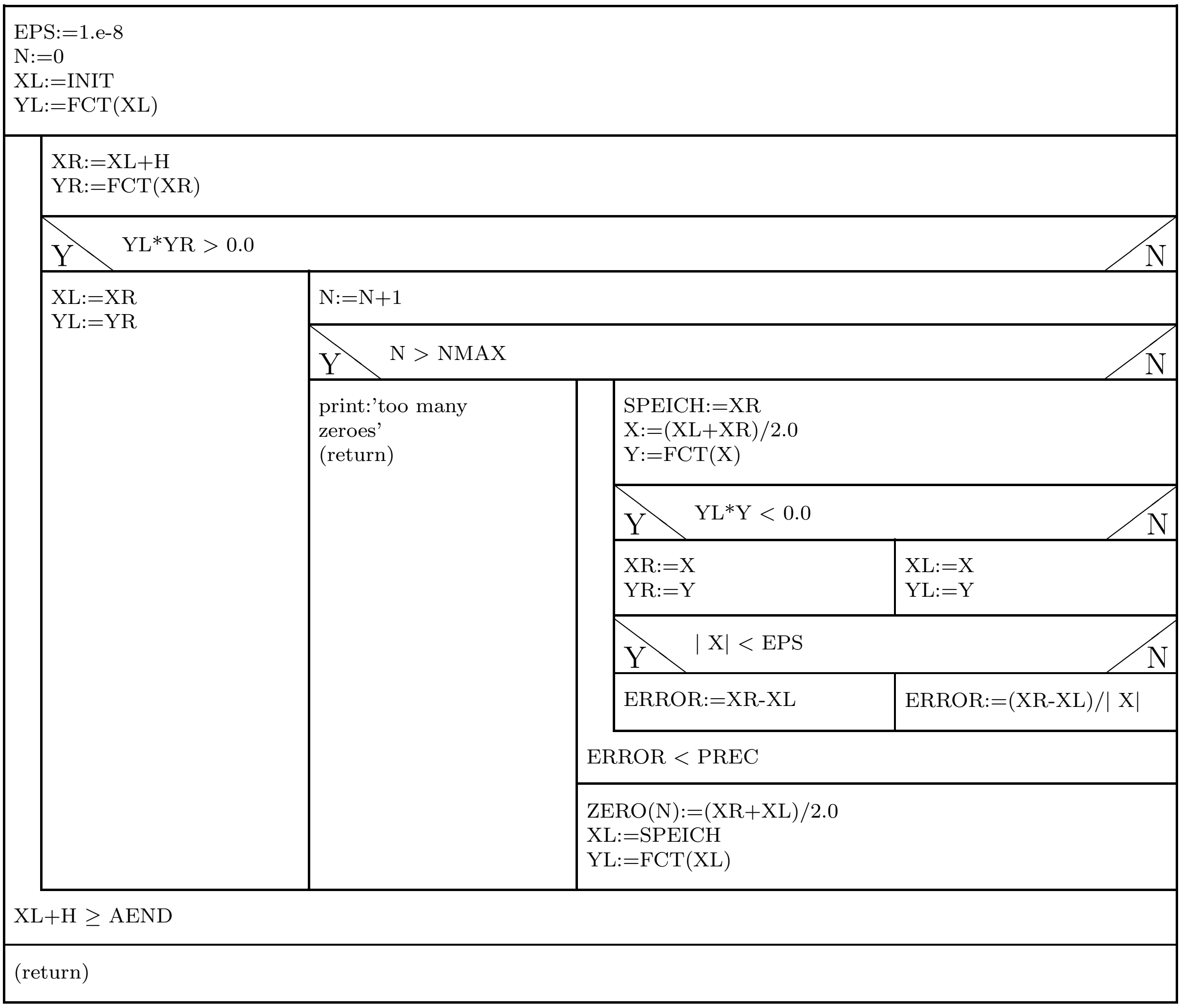
General remark:

The program presented in the following is simplified with respect to the original one. In particular, the present implementation does not consider the special case in which a zero lies *exactly* on one of the two extrema of the interval.

Remark on the error estimate:

The program INTSCH usually performs a relative error estimate. Only if the zero is very small (absolute value smaller than 10^{-8}) the program switches to an absolute error estimate.

Structure chart15 — INTSCH(INIT,AEND,H,PREC,NMAX,ZERO,N)



An example for INTSCH.

We wish to determine the real zeroes of the algebraic equation

$$F(x) = x^4 - 9x^3 - 2x^2 + 120x - 130 = 0$$

in the interval –10.0 to +10.0 with a relative precision of PREC=10^{-7}; the interval width of the gross search is 0.5.

INTSCH returns the four following zeroes:

```
1         -3.600135
2          1.228589
3          3.972068
4          7.399477
```

This result we obtain obviously is exactly the same as the Newton iteration (see section 4.3.3). However, it is interesting to compare the computational cost of the two methods.
If we count the number of calls to the subroutine 'func' in the Newton-Raphson program, we obtain 66; the same analysis applied to the bissection program returns 131 calls of 'fct'. However, since in the Newton method each call of 'func' actually means evaluating two functions, i.e. the function and its first derivative, we can conclude that the computational cost of the two programs is comparable.

This analysis however gives a wrong impression of the performance of the Newton-Raphson method. A large part of the function calls to 'func' occur during the gross search, where the calculation of the first derivative is not needed! It would therefore be more economical to define two independent functions (i.e. 'func' and 'dfunc'), that calculate respectively only the function and its first derivative.
If we proceed in this way, in the test problem we find 66 calls of 'func' and only 15 calls of 'dfunc', i.e. a total of 81 function calls.

In conclusion: The more economical the initial gross search, the stronger is the computational advantage of the Newton iteration compared to the bissection method. Consider for example the first zero: it lies in the 'gross search interval' [-4.0, -3.5]. In the following you see the subsequent steps needed to approximate the zero, with the Newton-Raphson method (left) and with the bissection (right):

```
Newton-Raphson    bissection:
-3.750000            -3.750000
-3.609011            -3.625000
-3.600169            -3.562500
-3.600135            -3.593750
-3.600135            -3.609375
                     -3.601562
                     -3.597656
                     -3.599609
                     -3.600586
                     -3.600098
                     -3.600342
                     -3.600220
                     -3.600159
                     -3.600128
                     -3.600143
                     -3.600136
                     -3.600132
                     -3.600134
                     -3.600135
                     -3.600135
                     -3.600135
                     -3.600135
```

An obvious advantage of the bissection method compared to the Newton-Raphson method is that it does not require any calculation of the first derivative of $F(x)$! This advantage is also found in the Regula Falsi method which has been briefly described in section 4.4.

4.5.3 An example from quantum mechanics.

In the following example we will calculate the energy eigenvalues of a one-dimensional potential well:

We want to determine which values E are allowed for the energy of a particle of mass m subject to a in a *square* potential well (see Fig..4.10):

$$V(x) = \begin{array}{cccc} 0 & \text{für} & -\infty < x \leq 0 & \text{Interval I} \\ -V_0 & \text{für} & 0 < x \leq a & \text{Interval II} \\ 0 & \text{für} & a < x < \infty & \text{Interval III} \end{array}$$

We consider only values of E which correspond to *bound states*:

$$-V_0 < E < 0$$

We thus have to solve a one-dimensional Schrödinger equation

$$-\frac{\hbar^2}{2m}\frac{d^2}{dx^2}\psi(x) + V(x)\psi(x) = E\psi(x) \quad .$$

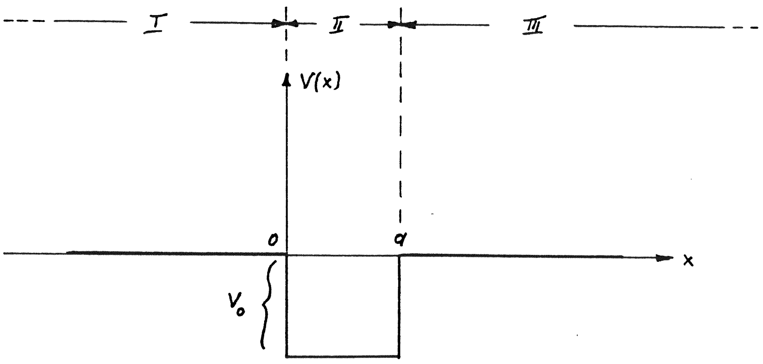


Figure 4.10: The one-dimensional potential well.

In atomic units, i.e. lengths in Bohr and energies in Rydberg, $\hbar^2/2m \equiv 1$, our differential equation reads:

$$\psi''(x) + [E - V(x)]\psi(x) = 0$$

with the boundary conditions

$$\psi(+\infty) \to 0 \qquad \text{und} \qquad \psi(-\infty) \to 0 \quad .$$

<u>*Ansatz* for the solution:</u>

- For the intervals I and III:

$$\psi''(x) + E \cdot \psi(x) = 0 \quad \to \quad \begin{array}{c} \psi_I(x) = A_I e^{\sqrt{-E}x} + B_I e^{-\sqrt{-E}x} \\ \psi_{III}(x) = A_{III} e^{\sqrt{-E}x} + B_{III} e^{-\sqrt{-E}x} \end{array}$$

  The boundary conditions are satisfied only for $B_I = A_{III} = 0$, and we thus obtain:

$$\psi_I(x) = A_I e^{\sqrt{-E}x} \qquad \text{und} \qquad \psi_{III}(x) = B_{III} e^{-\sqrt{-E}x} \tag{4.19}$$

- Interval II:

$$\psi''(x) + [E + V_0]\psi(x) = 0 \quad \to$$

$$\psi_{II}(x) = A_{II} \sin\left(x\sqrt{E + V_0}\right) + B_{II} \cos\left(x\sqrt{E + V_0}\right) \tag{4.20}$$

Furthermore the solution has to satisfy the 4 *matching conditions*:

$$\begin{aligned} \psi_I(0) &= \psi_{II}(0) \\ \psi'_I(0) &= \psi'_{II}(0) \\ \psi_{II}(a) &= \psi_{III}(a) \\ \psi'_{II}(a) &= \psi'_{III}(a). \end{aligned} \tag{4.21}$$

If we insert (4.19) and (4.20) into (4.21), we obtain a set of four linear, homogeneous equations for the unknown coefficients A_I, A_{II}, B_{II} and B_{III}, (with the substituion $\sqrt{E+V_0} \equiv \kappa$).

$$\begin{aligned}
A_I - B_{II} &= 0 \\
A_I\sqrt{-E} - A_{II}\kappa &= 0 \\
A_{II}\sin\kappa a + B_{II}\cos\kappa a - B_{III}e^{-\sqrt{-E}a} &= 0 \\
A_{II}\kappa\cos\kappa a - B_{II}\kappa\sin\kappa a + B_{III}\sqrt{-E}e^{-\sqrt{-E}a} &= 0
\end{aligned}$$

In matrix language this system has the form

$$\underbrace{\begin{pmatrix} 1 & 0 & -1 & 0 \\ \sqrt{-E} & -\kappa & 0 & 0 \\ 0 & \sin\kappa a & \cos\kappa a & -e^{-\sqrt{-E}a} \\ 0 & \kappa\cos\kappa a & -\kappa\sin\kappa a & \sqrt{-E}e^{-\sqrt{-E}a} \end{pmatrix}}_{M(E)} \begin{pmatrix} A_I \\ A_{II} \\ B_{II} \\ B_{III} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{4.22}$$

The solution of the system (4.22) is different from the zero vector, if the determinant of the coefficient matrix $M(E)$ vanishes, i.e. the particle energy can have only those values, for which:

$$\det(M(E)) = 0$$

Calculating the determinant analytically is trivial, and we obtain the expression:

$$\det(M(E)) = -e^{-a\sqrt{-E}}\left[(V_0+2E)\sin\left(a\sqrt{E+V_0}\right) - 2\sqrt{-E(E+V_0)}\cos\left(a\sqrt{E+V_0}\right)\right]$$

Given that $\exp(-a\sqrt{-E})$ is always positive, *the energy eigenvalues of the problem are the real solutions of the equation*

$$F(E) = (V_0+2E)\sin\left(a\sqrt{E+V_0}\right) - 2\sqrt{-E(E+V_0)}\cos\left(a\sqrt{E+V_0}\right)$$

We will now try to determine the zeroes of this expression for arbitrary values of a and V_0 using the bissection method. This method is well suited in the present case, since in this way we can avoid calculating the derivative of $F(E)$.

Results for a potential well with $a = 2$ Bohr and $V_0 = 225$ Rydbergs:

We obtain 10 energy values with the relative precision of 0.000001:

```
     1    Energy [Ry] = -222.83185
     2    Energy [Ry] = -216.33258
     3    Energy [Ry] = -205.51910
     4    Energy [Ry] = -190.42145
     5    Energy [Ry] = -171.08820
     6    Energy [Ry] = -147.59515
     7    Energy [Ry] = -120.06418
     8    Energy [Ry] =  -88.70779
     9    Energy [Ry] =  -53.96208
    10    Energy [Ry] =  -17.15278
```

4.6 Non linear systems.

Problems of this form are discussed here only schematically, since, as we will now show, they can be solved numerically using the thoroughly discussed Gauss-Newton-Marquardt method.

If instead of a single equation $F(x) = 0$ we have a *system of n non linear equations* with n unknowns x_1,x_2,. . ., x_n, i.e.

$$\begin{aligned} F_1(x_1, x_2, \ldots, x_n) &= 0 \\ &. \\ &. \\ F_k(x_1, x_2, \ldots, x_n) &= 0 \\ &. \\ &. \\ F_n(x_1, x_2, \ldots, x_n) &= 0 \quad , \end{aligned} \tag{4.23}$$

these equations can be simultaneously satisfied, only if:

$$S = \sum_{k=1}^{n} [F_k(x_1, x_2, \ldots, x_n)]^2 = 0$$

In this case therefore the zero we look for is the minimum of the sum S, and we look for one or more vectors $\mathbf{x}$ that satisfy

$$S = \sum_{k=1}^{n} [F_k(\mathbf{x})]^2 \to \text{Minimum} \tag{4.24}$$

The problem (4.23) is thus recast into a *special non-linear least-squares problem*, which can be solved with the Gauss-Newton-Marquardt method explained in the previous chapter:

- We first linearize the model through a Taylor series expansion of the functions $F_k(\mathbf{x})$ around $\mathbf{x} = \mathbf{x}^o$ ("initial vector"):

$$F_k(\mathbf{x}) = F_k(\mathbf{x}^o) + \sum_{j=1}^{n} \left(\frac{\partial F_k(\mathbf{x})}{\partial x_j} \right)_{\mathbf{x}^o} \cdot (x_j - x_j^o) + \cdots \quad . \tag{4.25}$$

We define:

$$dF_{k,j} \equiv \left(\frac{\partial F_k(\mathbf{x})}{\partial x_j}\right)_{\mathbf{x}^o} \qquad \text{und} \qquad F_k \equiv F_k(\mathbf{x}^o).$$

- We then insert the Taylor series expansions into (4.24) and calculate the partial derivative of the sum with respect to the unknowns x_1 to x_n:

$$S \approx \sum_{k=1}^{n} \left(F_k + \sum_{j=1}^{n} dF_{k,j} \cdot (x_j - x_j^o) \right)^2$$

$$\frac{\partial S}{\partial x_l} = 2 \sum_{k=1}^{n} \left(F_k + \sum_{j=1}^{n} dF_{k,j} \cdot (x_j - x_j^o) \right) \cdot F_{k,l} = 0$$

for all $l = 1, \ldots, n$.

- From this we obtain the following linear system of equations for a *iterative improvement* of the solutions:

$$A \cdot (\mathbf{x} - \mathbf{x}^o) = \beta \qquad \text{with}$$

$$A = [\alpha_{ij}] \qquad \alpha_{ij} = \sum_{k=1}^{n} dF_{k,i} \cdot dF_{k,j} \qquad \text{und} \qquad \beta_i = -\sum_{k=1}^{n} F_k \cdot dF_{k,i} \tag{4.26}$$

If we now compare the system (4.26) with the system (3.20), the close relation between the two is evident. Furthermore, for the solution of (4.26) it is advisable to use the 'Marquardt variant', which ensures a better convergence of the iteration.

4.6.1 An example.

We want to solve the system [1]

$$x^3 - 3xy^2 - 1 = 0 \qquad y^3 - 3x^2y = 0$$

with the method described in section 4.6.

Exact Solutions:
$(x = 1,\, y = 0) \qquad (x = -1/2,\, y = \sqrt{3}/2) \qquad (x = -1/2,\, y = -\sqrt{3}/2)$

Numerical solution with the Gauss-Newton-Marquardt method:

```
.
absolute precision of the fitted value =.100000E-06


tmax =    50

guessed values:   x(1) = -.2000000E+00
                  x(2) = -.5000000E+00

  0    .740389E+00     -.200000E+00     -.500000E+00
*
*
*
*
*
  1    .100543E+00     -.595187E+00     -.823118E+00
  2    .374490E-02     -.513359E+00     -.850427E+00
  3    .478015E-05     -.500104E+00     -.865304E+00
  4    .227994E-09     -.500000E+00     -.866020E+00
  5    .543610E-15     -.500000E+00     -.866025E+00
  6    .543610E-15     -.500000E+00     -.866025E+00

Results through MRQMIN and MRQCOF:
=====================================
  x(1) = -.50000E+00      (exact: -0.5000000)
  x(2) = -.86603E+00      (exact: -0.8660254)
```

Remark: As it is customary in iterative methods in the application of non linear problems, which of the three solutions is obtained depends on the starting value of the GNM process. With the starting values chosen above one obtains the third solution.

In order to obtain the first solution we could use, for example, $x(1)$=–1.0 and $x(2)$=0.0 as starting values; for the second solution, $x(1)$=–1.0 und $x(2)$=1.0.

[1] from: F. Stummel, K. Hainer, *Praktische Mathematik*, Teubner Studienbücher, Stuttgart 1971.

Chapter 5

Eigenvalues and Eigenvectors of real matrices

5.1 Introduction: general and regular eigenvalue problems.

In chapter 2 we have considered *inhomogeneous* problems of the form:

$$A \cdot \mathbf{x} = \mathbf{f} \qquad A\text{: real matrix}$$

In the following, we will treat *homogeneous* problems, *i.e.* problems for which $\mathbf{f}$ is the *zero* vector:

$$A \cdot \mathbf{x} = 0 \tag{5.1}$$

According to theory, we have to distinguish two cases:

- The determinant of the matrix of coefficients A is different from zero: in this case the solution $\mathbf{x}$ is the zero vector.
- The determinant of A is zero: in this case, apart from the *trivial solution* $\mathbf{x} = 0$, there is also a non-trivial one $\mathbf{x} \neq 0$. For example, the homogeneous system:

  $$\begin{pmatrix} 1 & 2 & 3 \\ -1 & 13 & 2 \\ 0 & 3 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = 0$$

  admits the non-trivial solution $\mathbf{x} = (-7, -1, +3)$. It is clear that this solution is defined only up to a multiplicative constant c: in fact, every vector $c \cdot \mathbf{x}$ is still a solution of the system.

In the following, however, we do not want to consider systems of type (5.1). In fact, the most important systems for practical applications are homogeneous systems of the type

$$A(\lambda) \cdot \mathbf{x} = 0 \qquad \text{with} \qquad A(\lambda) = [a_{ij}(\lambda)] \quad , \tag{5.2}$$

in which the elements of the matrix A depend on a variable λ.
The solutions of (5.2) are:

- the trivial solution $\mathbf{x} = 0$.
- non-trivial solutions $\mathbf{x}_1, \mathbf{x}_2, \ldots$, when λ assumes one of the values $\lambda_1, \lambda_2, \ldots$, for which the matrix A is *singular*.

The *condition for non-trivial solutions* is thus:

$$det[A(\lambda)] = 0 \tag{5.3}$$

The (complex) solutions of equation (5.3), $\lambda_1, \lambda_2, \ldots$, and the corresponding vectors $\mathbf{x}_1, \mathbf{x}_2, \ldots$ are the *eigenvalues* and *eigenvectors* of the system (5.2). The determination of the eigenvalues and eigenvectors of (5.2) is called the solution of the *generalized eigenvalue problem.* Usually, there is essentially only one possibility to solve this kind of problems, i.e. determine the roots of eq. (5.3) numerically. This requires calculating many determinants and, therefore, this method is numerically very expensive.

A very important sub-case of the generalized eigenvalue problem (5.2) is that of *regular eigenvalue problems*, with

$$A(\lambda) = A_0 - \lambda \cdot I \qquad \text{I ... Identity matrix} \quad , \tag{5.4}$$

Note that the matrix A_0 does not depend on λ! These problems admit more efficient methods of solution.

For regular eigenvalue problems

$$(A_0 - \lambda I)\mathbf{x} = 0 \qquad \text{i.e.} \qquad A_0\,\mathbf{x} = \lambda\,\mathbf{x} \tag{5.5}$$

the condition for non-trivial solution reads:

$$det(A_0 - \lambda I) = 0 \quad . \tag{5.6}$$

In the following, we will consider only regular eigenvalue problems, and, for brevity, omit the subscript 0 in A_0.

The evaluation of (5.6) leads to a n-th degree polynomial in λ, where n is the order of the linear system (5.5). This polynomial is called the *characteristic polynomial of the matrix* A:

$$det(A - \lambda I) \equiv P_n(\lambda) = \lambda^n + \sum_{i=1}^{n} p_i \lambda^{n-i} \tag{5.7}$$

The $p_1, \ldots, p_n$ are the (<u>real</u>) coefficients of the characteristic polynomial. This polynomial has exactly n (not necessarily different) zeroes, which are either real or complex-conjugate. Due to (5.6), the n zeroes are the eigenvalues of the matrix A. From this it immediately follows that the characteristic polynomial has *multiple zeroes*, i.e. that it holds:

$$P_n(\lambda) = 0 \qquad \text{for} \qquad \lambda : \lambda_1, \lambda_2, \ldots, \lambda_k = \lambda_{k+1}, \ldots = \lambda_{k+q-1}, \lambda_{k+q}, \ldots, \lambda_n \quad .$$

The eigenvalue λ_k is q *times degenerate.*

According to (5.5), we can associate an eigenvector $\mathbf{x}_i$ to each eigenvalue λ_i.

5.1.1 Eigenvalues and eigenvectors of matrix with special forms

Just like in the case of *inhomogeneous* linear systems, also for eigenvalue problems the concrete form of A plays an important role. In particular:

- A *real* matrix A is called *symmetrical*, if
$$A = A^T \,,$$
where A^T is the transpose of matrix A, with components
$$(a_{i,j})^T = a_{j,i}.$$
- A *complex* matrix A is called *hermitian* if:
$$A = A^\dagger \,,$$
where $A^\dagger$ is the hermitian conjugate matrix of A with components
$$(a_{i,j})^\dagger = a^*_{j,i}.$$

All the eigenvalues of a symmetricalor hermitian matrix are real.

- A real matrix A is called *orthogonal*, if:
$$A\,A^T = I \qquad \text{i. e.} \qquad A^T = A^{-1}\,,$$
where I is the identity matrix.
- A complex matrix A is called *unitary* if
$$A\,A^\dagger = I \qquad \text{i. e.} \qquad A^\dagger = A^{-1}\,.$$
- A real or complex matrix is called *normal*, if it commutes with its transpose or hermitian conjugate:
$$A\,A^T = A^T\,A \qquad \text{or} \qquad A\,A^\dagger = A^\dagger\,A\,.$$
It is obvious that both symmetrical and hermitian matrices are normal matrices.

A very important question is also to know whether a given matrix is *diagonalizable.*
Diagonalizable means that there exists a non-singular matrix U which can transform A into a diagonal matrix D:
$$U^{-1}\,A\,U = D\,. \tag{5.8}$$
This type of transformation is a *similarity operation*, which does not change the spectrum of the matrix; this means that:
$$\text{Eigenvalues of } D = \text{Eigenvalues of } A\,.$$

- Since the eigenvalue problem of D is easy to solve (the eigenvalues diagonal elements of D), the diagonalisation operation (5.8) is equivalent to the determination of the eigenvalues of A:

$$\lambda_i = d_{ii} \,. \tag{5.9}$$

- We can easily show that the column vectors of U coincide with the eigenvectors of A:

  We multiply the transformation (5.8) from left by U, and obtain:

$$U \cdot U^{-1} A U = U \cdot D \qquad \text{and}$$

$$A \cdot U = U \cdot D \quad . \tag{5.10}$$

  The matrix U can also be represented as a *system of n column vectors*:

$$U = \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ u_{21} & u_{22} & \cdots & u_{2n} \\ . & . & & . \\ . & . & & . \\ u_{n1} & u_{n2} & \cdots & u_{nn} \end{pmatrix} \equiv (\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n) \quad .$$

  The component-by-component evaluation of the right side of (5.10) gives:

$$(UD)_{ij} = \sum_{l=1}^{n} u_{il} d_{lj} = u_{ij} d_{jj} = \lambda_j u_{ij} \quad .$$

  The matrix UD can also be written in terms of vectors using (5.9):

$$UD \equiv (\lambda_1 \mathbf{u}_1, \lambda_2 \mathbf{u}_2, \ldots, \lambda_n \mathbf{u}_n) \quad .$$

  We can thus rewrite (5.10) as:

$$A \cdot (\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n) = (\lambda_1 \mathbf{u}_1, \lambda_2 \mathbf{u}_2, \ldots, \lambda_n \mathbf{u}_n) \quad .$$

  i.e. each column vector $\mathbf{u}_j$ satisfies the eigenvalue condition, and we can write: *The j-th column of the transformation matrix U represents the j-th eigenvector of the original matrix A.*

In summary: If we are able to find the transformation matrix U we are able to solve the eigenvalue problem for the matrix A!

Which matrices are diagonalizable?

- All those matrices whose eigenvectors form *linearly independent* system.
- In particular it is possible to show that the eigenvectors of *normal* matrices have this property. Furthermore, the eigenvalues of these matrices are also *orthonormal*, i.e. they obey the relation

$$\mathbf{x}_i \cdot \mathbf{x}_j^* = \delta_{i,j} \,.$$

Since the eigenvectors of A are the column vectors of the transformation matrix U, it follows:

$$U\,U^T = I \qquad \text{and} \qquad U\,U^\dagger = I\,,$$

i.e. the matrices U are *orthogonal* and *unitary.*

Normal matrices have the following important property:
Normal matrices can be reduced into diagonal form through an orthogonal and unitary transformation:

$$U^T\,A\,U = D \qquad \text{and} \qquad U^\dagger\,A\,U = D\,. \tag{5.11}$$

5.2 Numerical solution of regular eigenvalue problems

5.2.1 General Considerations

The numerical solution of regular eigenvalue problem is a sub-field of numerical mathematics of extraordinary importance. Over the years, several very specialised methods have been developed. In order to choose the most suitable method for a given problem, the user must therefore know exactly in advance what his requirements are.

In the following I will illustrate the basic ideas of the most important approaches using example of simple, yet effective, methods. A long and exhaustive review of all possible methods is outside the scope of this lecture.

In the following we will restrict our analysis to *real* matrices and describe:

- the method of von Mises (method of vector iteration);
- the method of Jacobi;
- the application of the Hyman method to Hessenberg matrices.

5.3 The method of von Mises

This iterative method returns the eigenvalue which has the largest (or, as we will see in the following, the smallest) absolute value for a given real matrix A. [1]
In order to apply this method we require that the matrix A is diagonalizable (i.e. its eigenvectors must form a linearly independent system) and possesses one dominant eigenvalue:

$$\mid \lambda_1 \mid > \mid \lambda_2 \mid \geq \mid \lambda_3 \mid \geq \ldots \geq \mid \lambda_{n-1} \mid > \mid \lambda_n \mid \ . \tag{5.12}$$

[1] In the following, for brevity we will always talk about *smallest* and *largest* eigenvalue, meaning the eigenvalue with the smallest of largest absolute value.

If the eigenvectors of A are linearly independent, each other vector $\mathbf{v}^{(0)}$, with the exception of the zero vector, can be expressed as a linear combination of the eigenvectors:

$$\mathbf{v}^{(0)} = \sum_{i=1}^{n} \alpha_i \mathbf{x}_i \qquad \text{with} \qquad \mid \alpha_1 \mid + \mid \alpha_2 \mid + \ldots + \mid \alpha_n \mid \neq 0 \quad .$$

We then choose $\mathbf{v}^{(0)}$ as the starting vector for the following iteration:[2]
- $\mathbf{v}^{(0)}$ is multiplied from left with the matrix A:

$$A \cdot \mathbf{v}^{(0)} \equiv \mathbf{v}^{(1)} = \sum_{i=1}^{n} \alpha_i A \mathbf{x}_i = \sum_{i=1}^{n} \alpha_i \lambda_i \mathbf{x}_i \quad ,$$

where the last identity follows from the eigenvalue equation (5.5).
- Now the vector $\mathbf{v}^{(1)}$ is multiplied itself from left with matrix A (*continued vector multiplication*)

$$A \cdot \mathbf{v}^{(1)} \equiv \mathbf{v}^{(2)} = \sum_{i=1}^{n} \alpha_i \lambda_i A \mathbf{x}_i = \sum_{i=1}^{n} \alpha_i \lambda_i^2 \mathbf{x}_i \quad .$$

Repeating these steps we obtain a sequence of vectors $\mathbf{v}^{(0)}$, $\mathbf{v}^{(1)}$, ..., $\mathbf{v}^{(t)}$, $\mathbf{v}^{(t+1)}$, ... with

$$\mathbf{v}^{(t)} = \sum_{i=1}^{n} \alpha_i \lambda_i^t \mathbf{x}_i \qquad (t = 0, 1, \ldots) \quad . \tag{5.13}$$

Now if (5.12) holds, with further iterations the first terms of the sum in (5.13) will become more and more dominant as the powers of λ increases. For not small values of t it is thus a good approximation to write:

$$\mathbf{v}^{(t)} \approx \alpha_1 \lambda_1^t \mathbf{x}_1 \qquad \text{and} \qquad \mathbf{v}^{(t+1)} \approx \alpha_1 \lambda_1^{t+1} \mathbf{x}_1 \quad . \tag{5.14}$$

The above equations are vector equations, i.e. they have to be satisfied for each of the lth components. We have thus:

$$\frac{v_l^{(t+1)}}{v_l^{(t)}} \approx \lambda_1 \qquad \text{for all} \qquad l = 1, 2, \ldots, n \quad . \tag{5.15}$$

This means that, if the iteration converges, the ratio between the same component of two subsequent vectors tends to the largest eigenvalue:

$$\lim_{t \to \infty} \frac{v_l^{(t+1)}}{v_l^{(t)}} \to \lambda_1 \quad . \tag{5.16}$$

Therefore it follows from (5.14) that:

$$\lim_{t \to \infty} \mathbf{v}^{(t)} \to \quad \text{prop.} \quad \mathbf{x}_1 \quad . \tag{5.17}$$

In principle we could choose an arbitrary l to evaluate equation (5.15). However, we cannot *a priori* exclude that during the iteration one of the $v_l^{(t)}$

[2] see however sect. 6.3.5.

becomes extremely small, or even zero. In order to avoid this, instead of (5.15) we evaluate the expression

$$\frac{1}{n'} \sum_{\mu} \frac{v_{\mu}^{(t+1)}}{v_{\mu}^{(t)}} \approx \lambda_1 \quad , \tag{5.18}$$

where we sum over all indices μ, for which:

$$|v_{\mu}^{(t)}| > \epsilon \,. \tag{5.19}$$

n' is the number of terms in the sum which satisfy this condition.

5.3.1 The calculation of the smallest eigenvector

of a real matrix is for many practical application more interesting than that of the largest eigenvalue. The method of von Mises can be applied also to this problem.

Starting from the eigenvalue equation (5.5)

$$A \cdot \mathbf{x}_i = \lambda_i \cdot \mathbf{x}_i$$

we multiply by the matrix inverse of A, A^{-1} and obtain:

$$\mathbf{x}_i = \lambda_i \cdot A^{-1} \mathbf{x}_i \quad .$$

The *eigenvalue equation for the inverse matrix* reads therefore

$$A^{-1} \cdot \mathbf{x}_i = \frac{1}{\lambda_i} \cdot \mathbf{x}_i \quad . \tag{5.20}$$

This means:

- The eigenvalues of a matrix A^{-1} are simply the inverse of the eigenvalues of A.
- The matrices A^{-1} and A have the same eigenvectors.

Since the inverse of the smallest eigenvectors of A are obviously the largest eigenvalues of A^{-1}, in order to determine the smallest possible eigenvalue of A we have to apply the von Mises method to the its inverse.

There is also an alternative way to obtain the smallest eigenvalue with the von Mises iteration. For this we again construct the same iteration procedure, but use the *inverse* of A:

$$\mathbf{v}^{(t+1)} = A^{-1} \, \mathbf{v}^{(t)} \,.$$

We then multiply this equation from left by A, and obtain

$$A \, \mathbf{v}^{(t+1)} = \mathbf{v}^{(t)} \,. \tag{5.21}$$

This can be interpreted as a linear system of equations with matrix of coefficients A, inhomogeneous vector $\mathbf{v}^{(t)}$ and the solution $\mathbf{v}^{(t+1)}$. If we apply

the LU decomposition, explained in chapter 2, to determine the solution of this system, we only need to compute the LU decomposition of A only once, (before the beginning of the Mises iteration), and after this use only the program LUBKSB for the following steps of the iteration.
In this way we obtain a series of vectors with the property

$$\lim_{t\to\infty} \frac{v_l^{(t)}}{v_l^{(t+1)}} = \lambda_n\,, \tag{5.22}$$

where λ_n is the smallest eigenvector of A.

5.3.2 Improvement of the convergence through spectral shift

Also without mathematical proof it is immediately evident that the convergence of the iteration (5.22) will be better the larger the ratio

$$\frac{1}{\lambda_n}/\frac{1}{\lambda_{n-1}} = \frac{\lambda_{n-1}}{\lambda_n}.$$

This can be exploited as follows:

Let us assume we know that λ_0 is a good approximation for the smallest eigenvalue λ_n. If instead of A we consider the matrix

$$A' = A - \lambda_0\, I\,,$$

all eigenvalues will be shifted by λ_0. The same holds for the eigenvalues λ_{n-1} and λ_n, and

$$\frac{\lambda_{n-1}-\lambda_0}{\lambda_n-\lambda_0} > \frac{\lambda_{n-1}}{\lambda_n}\,.$$

Therefore we expect that the limiting value

$$\lim_{t\to\infty} \frac{v_l^{(t)}}{v_l^{(t+1)}} + \lambda_0 = \lambda_n \tag{5.23}$$

will rapidly converge to the value (5.22).

The considerations in sections 6.3.1 and 6.3.2 are realised in the program MISES.

5.3.3 The program MISES

Source: [2], P.94f, program P. 331 f with modifications.

The program MISES calculates the smallest eigenvalue and the corresponding eigenvector of a real matrix (including the spectral shift).

INPUT parameters:

A(,): Real matrix whose smallest eigenvalue we want to determine.

N: Order of the matrix.

TMAX: Maximum number of iterations.

PREC: Relative precision of the eigenvalue.

V( ): Initial vector $\neq$ zero vector.

EIGW0: Estimate of the smallest eigenvalue.

OUTPUT parameters:

EIGW: Approximation for the smallest eigenvalue.

V( ): Normalised eigenvector corresponding to the eigenvalue λ_1.

ERROR: Logical error variable: TRUE if convergence is not reached in TMAX iteration, FALSE otherwise.

INTERNAL parameters:

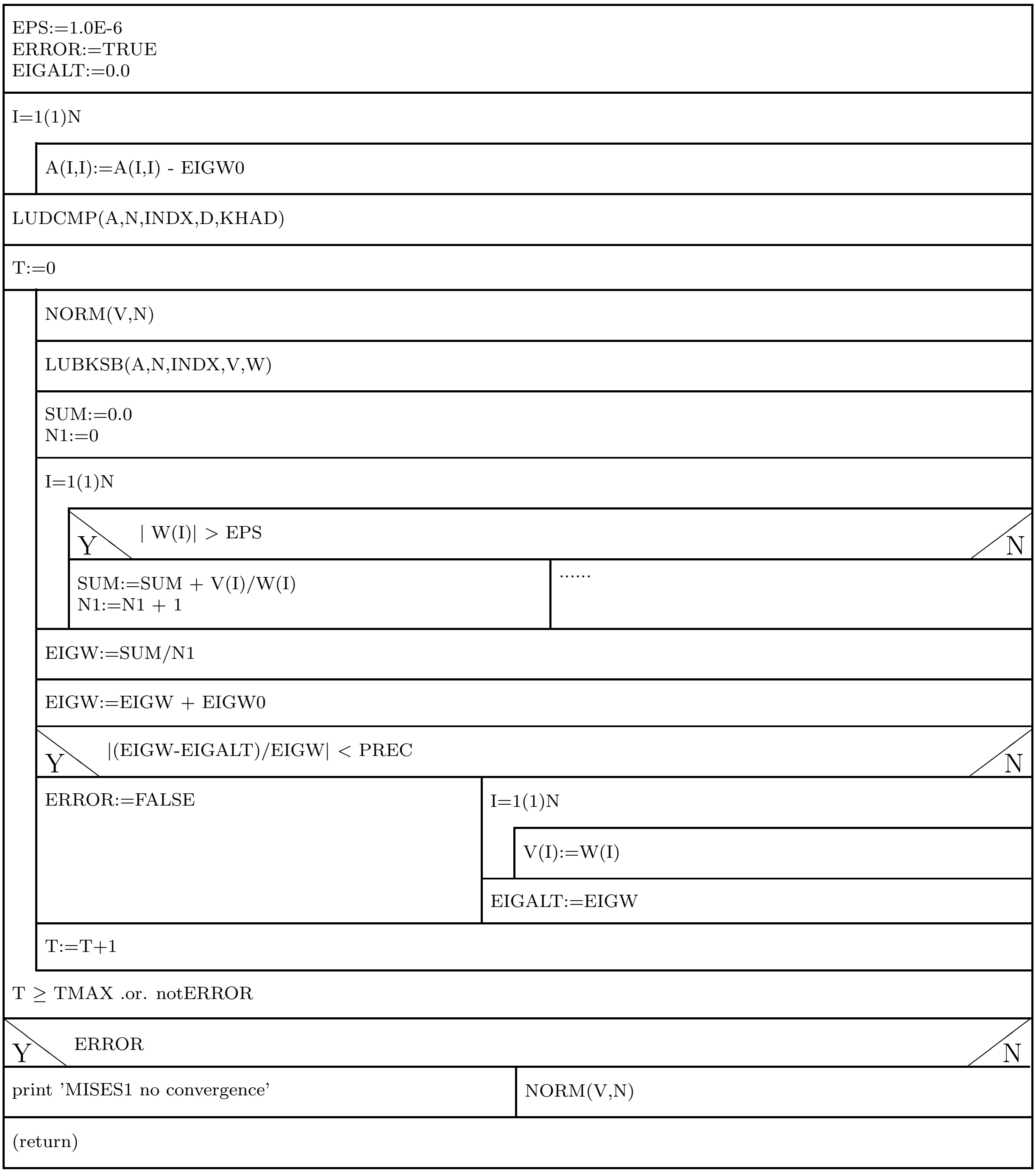
W: Auxiliary vector.

EPS: Constant, see Eq. (5.19).

Routines employed:

The program MISES requires the routines NORM, LUDCMP and LUBKSB (see chapter 2).

Structure chart 21 — MISES(A,N,TMAX,PREC,V,EIGW0,EIGW,ERROR)



```
EPS:=1.0E-6
ERROR:=TRUE
EIGALT:=0.0
I=1(1)N
    A(I,I):=A(I,I) - EIGW0
LUDCMP(A,N,INDX,D,KHAD)
T:=0
    NORM(V,N)
    LUBKSB(A,N,INDX,V,W)
    SUM:=0.0
    N1:=0
    I=1(1)N
        | W(I)| > EPS
        Y:
            SUM:=SUM + V(I)/W(I)
            N1:=N1 + 1
        N:
            ......
    EIGW:=SUM/N1
    EIGW:=EIGW + EIGW0
    |(EIGW-EIGALT)/EIGW| < PREC
    Y:
        ERROR:=FALSE
    N:
        I=1(1)N
            V(I):=W(I)
        EIGALT:=EIGW
    T:=T+1
T ≥ TMAX .or. notERROR
ERROR
Y:
    print 'MISES1 no convergence'
N:
    NORM(V,N)
(return)
```

Structure chart 21a — NORM(V,N)

```
SUM:=0.0
I=1(1)N
    SUM:=SUM + V(I)*V(I)
SUM:=SQRT(SUM)
SUM ≠ 0.0
  Y:
    I=1(1)N
        V(I):=V(I)/SUM
  N:
    print 'NORM zero vector'
(return)
```

5.3.4 Von Mises method: Tests and problems.

With the following examples we wish to illustrate how stable the program MISES is.

The tests start with the 4x4 matrix

```
3.8000    1.8000   -2.0000   -0.6000
5.4000    6.2000   -7.2000   -1.0000
2.0000    2.4000   -2.0000    0.0000
1.8000    1.0000    0.0000    1.0000
```

We wish to evaluate the smallest eigenvalue and the corresponding eigenvector. The exact result reads

$$\lambda_4 = 0.6 \qquad \text{and} \qquad \mathbf{x}_4 = \frac{1}{\sqrt{23}} \begin{pmatrix} 1 \\ -3 \\ -2 \\ 3 \end{pmatrix} = \begin{pmatrix} 0.2085144 \\ -0.6255432 \\ -0.4170288 \\ 0.6255432 \end{pmatrix} .$$

We now want to show how the spectral shift (see section 6.3.3) works. The first test worked *without* this technique, i.e. the starting (guess) value for λ_4 is set to zero:

```
.
MISES-test:   n =    4   rel. prec. =  1.0000000000E-06

Estimate value for EW =   0.000000

Matrix and initial vector:
    3.8000    1.8000   -2.0000   -0.6000        1.0000
    5.4000    6.2000   -7.2000   -1.0000        1.0000
    2.0000    2.4000   -2.0000    0.0000        1.0000
    1.8000    1.0000    0.0000    1.0000        1.0000

Smallest eigenvalue after 20 iterations             =   0.600000

                 Eigenvector:
       1     -0.208514
       2      0.625543
       3      0.417029
       4     -0.625543
```

If we now take as guess value $\lambda_0 = 0.5$, the program MISES1 returns:

```
MISES test:   n =    4   rel. prec. =  1.0000000000E-06

Estimate for EW =   0.500000

Matrix and initial vector:

    3.8000    1.8000   -2.0000   -0.6000        1.0000
    5.4000    6.2000   -7.2000   -1.0000        1.0000
    2.0000    2.4000   -2.0000    0.0000        1.0000
    1.8000    1.0000    0.0000    1.0000        1.0000

Smallest eigenvalue after 8 iterations             =   0.600000

                 Eigenvector:
       1     -0.208514
       2      0.625543
       3      0.417029
       4     -0.625543
```

Note:
As you can see, MISES1 does not return the eigenvector predicted by theory, but the same eigenvector times (–1). Is there an error in the program?

Finally, one more test for MISES on the 3x3 matrix

```
    1.0000    0.0000   -1.0000
    1.0000    2.0000    1.0000
   -2.0000   -2.0000    2.0000
```

with the two eigenvalues λ_1 and λ_2. The third (and smallest) eigenvalue and its eigenvector read:

$$\lambda_3 = 1.0 \qquad \text{and} \qquad \mathbf{x}_3 = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0.7071068 \\ -0.7071068 \\ 0.0000000 \end{pmatrix} .$$

```
.
MISES test:   n =    3   rel. prec. =  1.0000000000E-06

Estimate for EW =   0.000000

Matrix und initial vector:

    1.0000    0.0000   -1.0000        1.0000
    1.0000    2.0000    1.0000        0.0000
   -2.0000   -2.0000    2.0000        0.0000

smallest eigenvalue after 24 iterations =   1.000000

                 Eigenvector:
       1      0.707107
       2     -0.707107
       3      0.000000
```

5.4 The method of Jacobi

This method permits to *solve the full eigenvalue problem for symmetrical matrices.*
Let us consider a real matrix

$$A = [a_{ij}] \qquad \text{with} \qquad a_{ij} = a_{ji} \quad .$$

It is a general property of real symmetrical matrices to have *only real eigenvalues.*

The Jacobi method is based on an orthogonal transformation of the matrix A into a diagonal matrix D (see Chap. 6.1.1):

$$U^T \cdot A \cdot U = D \tag{5.24}$$

As already extensively discussed, once the matrix U is known, the eigenvalue problem for A is completely solved:

- The diagonal elements of D are the eingenvalues of A:

$$\lambda_i = d_{ii} \qquad (i = 1, \ldots, n)\,.$$

- The columns of the transformation matrix U are the eigenvectors of A (up to a renormalization factor).

5.4.1 An iterative approximation for the tranformation matrix U

We now have the problem of finding the transformation matrix U. In the Jacobi method the transformation (5.24) is approximated by a $\overline{\text{sequence}}$ of similarity operations. Obviously the transformation matrices $U_t \quad (t = 0, 1, 2, \ldots)$ have to be orthogonal.

Instead of (5.24) we have:

$$\begin{gathered}
U_0^T A U_0 \equiv A^{(1)} \\
U_1^T A^{(1)} U_1 \equiv A^{(2)} = U_1^T U_0^T A U_0 U_1 \\
. \\
. \\
U_t^T A^{(t)} U_t \equiv A^{(t+1)} = U_t^T U_{t-1}^T \cdots U_0^T A U_0 \cdots U_{t-1} U_t \,,
\end{gathered}$$

where the single transformations have to be chosen so that the the sequence of matrices converges to:

$$\begin{aligned}
\lim_{t\to\infty} A^{(t)} \quad &\to \quad D \\
\lim_{t\to\infty} U_0 U_1 \cdots U_{t-1} U_t \quad &\to \quad U
\end{aligned}$$

The matrices U_t used in the Jacobi method have the general form of a *orthogonal rotation matrices.*

$$. \tag{5.25}$$

$$U_t(i,j,\varphi) = \begin{array}{ccccccl}
1 & & \vdots & & \vdots & & \\
 & 1 \ddots & \vdots & & \vdots & & \\
\cdots & \cdots & \cos\varphi & \cdots & -\sin\varphi & \cdots & i\text{-te} \\
 & & \vdots & \ddots\ 1\ \ddots & \vdots & & \text{Zeile} \\
\cdots & \cdots & \sin\varphi & \cdots & \cos\varphi & \cdots & j\text{-te} \\
 & & \vdots & & \vdots & \ddots\ 1 & \\
 & & i\text{-te} & & j\text{-te Spalte} & &
\end{array}$$

As we immediately see, U_t is specified by three conditions:
i and j indicate the rows and columns of the matrix for which the rotation matrix differs from the identity matrix. φ is the rotation angle. It is easy to show that $U(i,j,\varphi)$ is always orthogonal, independently of the arguments, i.e.

$$U_t^T(i,j,\varphi) \cdot U_t(i,j,\varphi) = I$$

Before discussing the optimal choice of the parameters i, j, φ for each iteration, we analyse in detail the transformation:

$$U^T(i_{t-1}, j_{t-1}, \varphi_{t-1}) \cdot A^{(t-1)} \cdot U(i_{t-1}, j_{t-1}, \varphi_{t-1}) \quad \rightarrow \quad A^{(t)} \tag{5.26}$$

If we rewrite it component by component we obtain the following expression:

$$a_{kl}^{(t)} = \sum_{m=1}^{n} \sum_{m'=1}^{n} u_{mk} u_{m'l} a_{mm'}^{(t-1)} \quad . \tag{5.27}$$

As easily seen in the above equation, the matrix $A^{(t-1)}$ retains its symmetry during the transformation; therefore we have

$$a_{kl}^{(t)} = a_{lk}^{(t)} \quad .$$

We now evaluate (5.27), for *k and l which are are neither i or j*. Under these conditions the components of the rotation matrix (5.25) are 'Kronecker deltas', and we obtain

$$a_{kl}^{(t)} = \sum_{m=1}^{n} \sum_{m'=1}^{n} \delta_{mk} \delta_{m'l} a_{mm'}^{(t-1)} = a_{kl}^{(t-1)} \quad .$$

In summary: *The transformation leaves all the components of A which do not contain the index i or j unchanged.*

The missing components of the i-th and j-th row and column can be determined from (5.27):

$$l = i \qquad k = 1, \ldots, n \quad \text{with} \quad k \neq i, j \quad :$$

$$a_{ki}^{(t)} = a_{ik}^{(t)} = a_{ki}^{(t-1)} \cos\varphi + a_{kj}^{(t-1)} \sin\varphi \tag{5.28}$$

$$l = j \qquad k = 1, \ldots, n \quad \text{with} \quad k \neq i, j \quad :$$

$$a_{kj}^{(t)} = a_{jk}^{(t)} = a_{kj}^{(t-1)} \cos\varphi - a_{ki}^{(t-1)} \sin\varphi \tag{5.29}$$

Finally for the components:

$$a_{ii}^{(t)} \qquad a_{jj}^{(t)} \qquad a_{ij}^{(t)} = a_{ji}^{(t)} \quad .$$

the transformations read:

$$a_{ii}^{(t)} = a_{ii}^{(t-1)} \cos^2 \varphi + 2a_{ij}^{(t-1)} \cos\varphi \sin\varphi + a_{jj}^{(t-1)} \sin^2 \varphi \tag{5.30}$$

$$a_{jj}^{(t)} = a_{jj}^{(t-1)} \cos^2 \varphi - 2a_{ij}^{(t-1)} \cos\varphi \sin\varphi + a_{ii}^{(t-1)} \sin^2 \varphi \tag{5.31}$$

$$a_{ij}^{(t)} = a_{ij}^{(t-1)} (\cos^2 \varphi - \sin^2 \varphi) + (a_{jj}^{(t-1)} - a_{ii}^{(t-1)}) \cos\varphi \sin\varphi \tag{5.32}$$

5.4.2 Choice of the parameters i, j and φ.

The aim of each of the orthogonal transformations applied to the matrix A is that of bringing the the corresponding matrices

$$A^{(1)}, A^{(2)}, \ldots, A^{(t-1)}, A^{(t)}, A^{(t+1)}, \ldots$$

closer and closer to a diagonal form, through iterations.

A good measure for the convergence clearly the *the sum S of the squares of the non-diagonal elements of the matrix*

$$S^{(t)} = 2 \sum_{m=1}^{n-1} \sum_{m'=m+1}^{n} \left(a_{mm'}^{(t)} \right)^2 \quad .$$

This means that if during a Jacobi iteration we perform the transformation

$$U_{t-1}^T \cdot A^{(t-1)} \cdot U_{t-1} \quad \rightarrow \quad A^{(t)}$$

we can consider it succesful if we have

$$S^{(t-1)} > S^{(t)} \quad ,$$

where $S^{(t-1)}$ and $S^{(t)}$ are the sums of the squares of the non-diagonal elements of the matrices $A^{(t-1)}$ and $A^{(t)}$.

Without proof: It is possible to show that the difference between $S^{(t-1)}$ and $S^{(t)}$ is given by

$$S^{(t-1)} - S^{(t)} = 2 \left[\left(a_{ij}^{(t-1)} \right)^2 - \left(a_{ij}^{(t)} \right)^2 \right] \tag{5.33}$$

i.e. it is given only by the matrix elements a_{ij} *before and after* the transformation.

The equation (5.33) suggests what the best strategy for the choice of the free parameters i, j und φ at a given iteration:
The decrease of S is stronger,

- *if the absolute value of $a_{ij}^{(t-1)}$ is as large as possible*
- *if the transformation reduces $a_{ij}^{(t)}$ to zero.*

Choosing i and j as the indexes of the largest off-diagonal elements of A before each iteration [3], the second condition can be satisfied through an appropriate choice of the rotation angle φ. Setting (5.32) to zero we obtain: [4]

$$\tan 2\varphi = \frac{2a_{ij}^{(t-1)}}{a_{ii}^{(t-1)} - a_{jj}^{(t-1)}} \quad . \tag{5.34}$$

For $a_{ii}^{(t-1)} = a_{jj}^{(t-1)}$ the rotation angle is:

$$\varphi = \frac{\pi}{4} \quad .$$

If now before each iteration we choose the parameters of the rotation matrix (5.25) as discussed above, we obtain a *monotonous* decrease of S

$$S^{(0)} > S^{(1)} > S^{(2)} > \cdots > S^{(t)} > \cdots$$

and this guarantees a faster approach to the required diagonal form.

Furthermore we still have to calculate the matrices which result from the product of the single transformation matrices:

$$B^{(t-1)} = U_0 U_1 U_2 \cdots U_{t-2} U_{t-1} \quad , \tag{5.35}$$

since this matrix contains approximately the eigenvectors of the matrix A, which we wish to determine. Once more it is possible to show that the multiplication of B with a rotation matrix $U_t(i, j, \varphi)$ changes only the elements on the i-th and j-th column of B:

$$\left(B^{(t-1)} U_t(i, j, \varphi)\right)_{k,i} = b_{ki} \cos\varphi + b_{kj} \sin\varphi \tag{5.36}$$

$$\left(B^{(t-1)} U_t(i, j, \varphi)\right)_{k,j} = b_{kj} \cos\varphi - b_{ki} \sin\varphi \tag{5.37}$$

for $k = 1, \ldots, n$.

[3] Indeed, finding the largest off-diagonal element before each matrix transformation is very time consuming. For this reason, most programs employ a simplified method, see sect. 6.4.3

[4] A variant of this method for C programs is discussed in Sect. 6.4.4

5.4.3 The program JACOBI.

The program JACOBI solves the full eigenvalue problem of a real, symmetric matrix.

INPUT parameters:

A(,): the components of a real, symmetrical matrix. Since the program JACOBI is formulated in such a way that it requires always only the matrix elements on and above the main diagonal, the program which calls JACOBI *needs to provide only these elements.*

N: Rows and columns of the matrix.

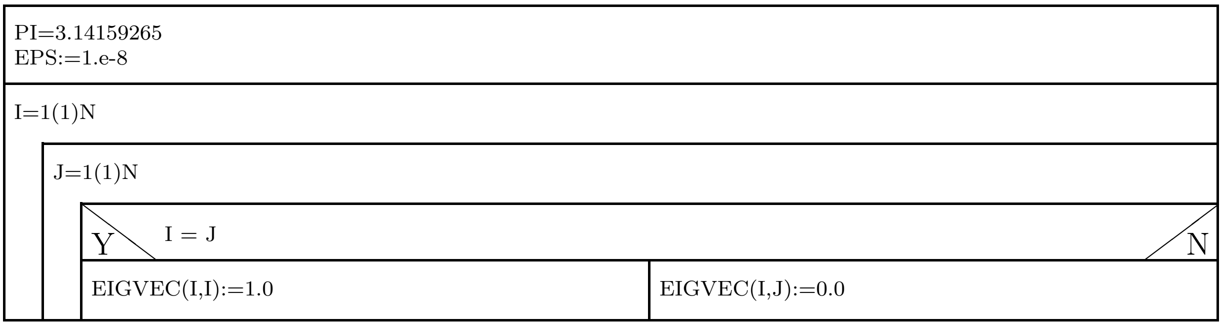
TMAX: maximum number of calculation steps.

OUTPUT parameters:

A(,): Approximation for a diagonal matrix with the same eigenvalue spectrum as the original matrix A. The diagonal elements of this matrix are approximate values for the eigenvalues.

EIGVEC(,): This matrix contains the eigenvectors of A, i.e. the j-th column of EIGVEC is the j-th eigenvector of A.

Structure chart 22 — JACOBI(A,N,TMAX,EIGVEC)



```
PI=3.14159265
EPS:=1.e-8
I=1(1)N
    J=1(1)N
        I = J
        Y: EIGVEC(I,I):=1.0
        N: EIGVEC(I,J):=0.0
```

Structure chart 23 — (Continuation)

```
T:=0
    T:=T+1
    SUM:=0.0
    I=1(1)N-1
        J=I+1(1)N
            SUM:=SUM+2.0*A(I,J)*A(I,J)
    IF(SUM=0.0) ===> (return)
    LIMIT:=SQRT(SUM)/N/N
    I=1(1)N-1
        J=I+1(1)N
            | A(I,J) | > LIMIT
            Y:
                DEN:=A(I,I)–A(J,J)
                | DEN/A(I,J) | < EPS
                Y:
                    PHI:=PI/4.0
                N:
                    PHI:=0.5*ATAN(2.0*A(I,J)
                    /DEN)
                SINUS:=SIN(PHI)
                COSIN:=COS(PHI)
                K=I+1(1)J-1
                    SAV:=A(I,K)
                    A(I,K):=COSIN*A(I,K)+SINUS*A(K,J)
                    A(K,J):=COSIN*A(K,J)–SINUS*SAV
                K=J+1(1)N
                    SAV:=A(I,K)
                    A(I,K):=COSIN*A(I,K)+SINUS*A(J,K)
                    A(J,K):=COSIN*A(J,K)–SINUS*SAV
                K=1(1)I-1
                    SAV:=A(K,I)
                    A(K,I):=COSIN*A(K,I)+SINUS*A(K,J)
                    A(K,J):=COSIN*A(K,J)–SINUS*SAV
                SAV:=A(I,I)
                A(I,I):=COSIN**2*A(I,I)+2.0*COSIN*SINUS*A(I,J)+SINUS**2*A(J,J)
                A(J,J):=COSIN**2*A(J,J)–
                2.0*COSIN*SINUS*A(I,J)+SINUS**2*SAV
                A(I,J):=0.0
                K=1(1)N
                    SAV:=EIGVEC(K,J)
                    EIGVEC(K,J):=COSIN*EIGVEC(K,J)–SINUS*EIGVEC(K,I)
                    EIGVEC(K,I):=COSIN*EIGVEC(K,I)+SINUS*SAV
            N:
                ......
T > TMAX
print 'JACOBI convergence not reached'
(return)
```

Structure of the program:

1. After the definition of the constant PI and of the pseudozero EPS=10^{-8} an identity matrix is stored into the array EIGVEC.

2. UNTIL loop for the (maximum) number of steps TMAX:

   - Determination of the sum of the squares of the non-diagonal matrix elements and of the threshold LIMIT.
   - Check whether the sum is zero.

     The convergence of the Jacobi method is most of the times so good that one can push the calculation up to the 'underflow' of SUM. The above query for the exact zero works only, if the system interterprets an underflow as zero!

     If SUM=0.0, return to the main program.

     If SUM>0.0, the matrix transformation according to (5.28 - 5.32) are performed with respect to *all off-diagonal elements with absolute values larger than LIMIT.*
   - Ater each transformation the matrix EIGVEC is multiplied by the orthogonal matrix U_t used in this iteration. (5.36 and 5.37).

3. If after TMAX calculation steps SUM is still larger than zero, JACOBI returns an ERROR message and returns to the main program.

5.4.4 Variant of the JACOBI algorithm in C.

In programs written in FORTRAN,PASCAL etc the calculation of the ideal rotation angle can be performed using Eq. (5.34).
In program languages which do not have an Arctangent function built in, such as for example C, the calculation of $\cos\varphi$ and $\sin\varphi$ can be carried out in the following way (for a mathematical derivation see [10], P. 464f):

```
.
  g=100*fabs(a[i][j]);
  if(fabs(a[i][j]) > limit) {
    h=a[i][i]-a[j][j];
// This if statement prevents an overflow of theta*theta in the following
// statement.
    if(fabs(h)+g == fabs(h)) tfac=a[i][j]/h;
    else {
      theta=h/2.0/a[i][j];
      tfac=1.0/(fabs(theta)+sqrt(1.0+theta*theta));
      if(theta<0.0)tfac=-tfac;
    }
    cosin=1.0/sqrt(tfac*tfac+1.0);
    sinus=tfac*cosin;
```

5.4.5 Two tests for JACOBI.

First Example:

```
.
Matrix:   5.000   4.000   1.000   1.000
          4.000   5.000   1.000   1.000
          1.000   1.000   4.000   2.000
          1.000   1.000   2.000   4.000

exact solution:

Eigenvalue:                    Eigenvector:

10  (2,2,1,1)/sqrt(10)   = (0.6324555, 0.6324555, 0.3162278, 0.3162278)
 1  (-1,1,0,0)/sqrt(2)   = (-0.7071068, 0.7071068, 0.0, 0.0)
 5  (-1,-1,2,2)/sqrt(10) = (-0.3162278, -0.3162278, 0.6324555, 0.6324555)
 2  (0,0,-1,1)/sqrt(2)   = (0.0, 0.0, -0.7071068, 0.7071068)

Jacobi-Solution:
Eigenvalue:                    Eigenvector:

10.000000       0.632456    0.632456    0.316228    0.316228
 1.000000      -0.707107    0.707107   -0.000000   -0.000000
 5.000000      -0.316228   -0.316228    0.632456    0.632456
 2.000000      -0.000000   -0.000000   -0.707107    0.707107
```

Second Example:

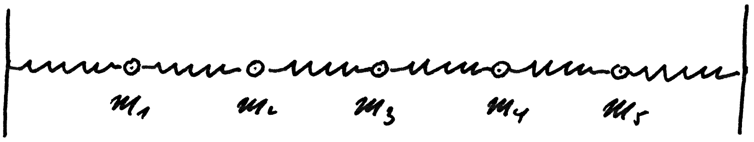
```
.
Matrix:   6.000   4.000   4.000   1.000
          4.000   6.000   1.000   4.000
          4.000   1.000   6.000   4.000
          1.000   4.000   4.000   6.000

exact solution:
Eigenvalue:                    Eigenvector:
15              0.5         0.5         0.5         0.5
-1              0.5        -0.5        -0.5         0.5
 5             -0.5         0.5        -0.5         0.5
 5             -0.5        -0.5         0.5         0.5

Jacobi Solution:
Eigenvalue:                    Eigenvector:

15.000000       0.500000    0.500000    0.500000    0.500000
-1.000000      -0.500000    0.500000    0.500000   -0.500000
 5.000000       0.226248   -0.669934    0.669934   -0.226248
 5.000000      -0.669934   -0.226248    0.226248    0.669934
```

Two further remarks on these results:

1. The normalized eigenvectors, obviously, are defined only up to a (multiplicative) constant (–1). Therefore, the fact that the program JACOBI returns the eigenvector corresponding to the eigenvalue -1 with the opposite sign is *not an ERROR*!
2. The effect of the non-uniqueness of the eigenvectors is even stronger in case of degenerate eigenvalues. In the case of the two times degenerate eigenvalue 5, shown above, this means that each linear combination of the two eigenvectors returned by JACOBI is still an eigenvector with eigenvalue 5. You can easily convince yourself that it is possible to find a linear combination of the two vectors $(-0.5, 0.5, -0.5, 0.5)$ and $(-0.5, -0.5, 0.5, 0.5)$ which corresponds to the eigenvectors found by JACOBI.

5.4.6 An application for the program JACOBI.

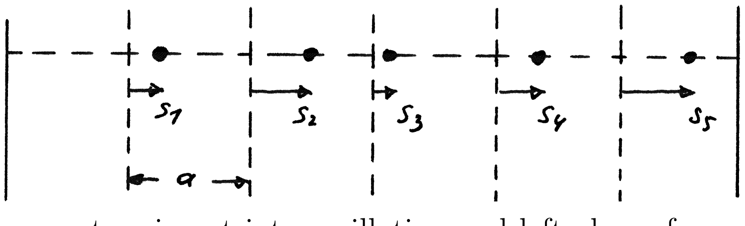
Let us consider a system of n point masses m_i, $i = 1, \ldots, n$ coupled by springs:



We will make the following assumptions:

- The restoring forces are so large, i.e. the masses so small, that the the gravitational force can be neglected.
- The oscillation amplitudes are small compared to the distance between the particles at rest.

In the following, a is the distance between two point masses at rest; s_i is the instantaneous displacement of the masses from their equilibrium position. We assume that the forces between the masses are proportional to their distance – *harmonic approximation.* The factor of proportionality between the distance and the force, the *force constant D*, is given in cgs-units (dyn/cm).



If such a system is put into oscillation and left alone, for each mass we can write an equation of the form

$$m_i \ddot{s}_i + D(a - s_{i-1} + s_i) - D(a - s_i + s_{i+1}) = 0$$

Here, m_i is the mass of the i-th particle and $s_i(t)$, $s_{i-1}(t)$, $s_{i+1}(t)$ are the instanteneous elongations of the i-th point and of its left and right neighbours. In the end, we obtain a *system of n coupled ordinary differential equations of second order*:

$$\begin{aligned} m_1\ddot{s}_1 + 2Ds_1 - Ds_2 &= 0 \\ m_i\ddot{s}_i - Ds_{i-1} + 2Ds_i - Ds_{i+1} &= 0 \qquad i = 2, 3, \ldots, n-1 \\ m_n\ddot{s}_n - Ds_{n-1} + 2Ds_n &= 0 \end{aligned} \tag{5.38}$$

If we restrict ourselves to small elongations, the system (5.38) can be reduced to a *homogeneous, linear system.* With the ansatz:

$$s_i(t) = \frac{b_i}{\sqrt{m_i}} \cdot e^{i\omega t}$$

we have:

$$\begin{aligned} \left(\frac{2D}{m_1} - \omega^2\right) b_1 - \frac{D}{\sqrt{m_1 m_2}} b_2 &= 0 \\ -\frac{D}{\sqrt{m_i m_{i-1}}} b_{i-1} + \left(\frac{2D}{m_i} - \omega^2\right) b_i - \frac{D}{\sqrt{m_i m_{i+1}}} b_{i+1} &= 0 \quad (i = 2, 3, \ldots, n) \\ -\frac{D}{\sqrt{m_n m_{n-1}}} b_{n-1} + \left(\frac{2D}{m_n} - \omega^2\right) b_n &= 0 \end{aligned}$$

which can be written as a *regular eigenvalue problem*:

$$. \tag{5.39}$$

$$\begin{pmatrix} \frac{2D}{m_1} & -\frac{D}{\sqrt{m_1 m_2}} & & \\ \ddots & \ddots & \ddots & \\ -\frac{D}{\sqrt{m_i \cdot m_{i-1}}} & \frac{2D}{m_i} & -\frac{D}{\sqrt{m_i \cdot m_{i+1}}} & \\ & \ddots & \ddots & \ddots \\ & & -\frac{D}{\sqrt{m_n \cdot m_{n-1}}} & \frac{2D}{m_n} \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ \vdots \\ b_i \\ \vdots \\ b_n \end{pmatrix} = \lambda \cdot \begin{pmatrix} b_1 \\ \vdots \\ b_i \\ \vdots \\ b_n \end{pmatrix}$$

with $\omega^2 = \lambda$. The (real) eigenvalues of the above *symmetrical, tridiagonal matrix* are thus the squares of the eigenfrequencies of the oscillating system (smallest frequency= zero harmonics plus $n-1$ over-tunes).

```
Test example "spring pendulum" for Jacobi:
=========================================

Spring constant [dyn/cm] = 25.000
5 masses:
  m1 = 3 g   m2 = 6 g   m3 = 9 g   m4 = 2 g   m5 = 6 g

JACOBI returns the following result:

   nr    lambda(1/s/s)     Frequency(1/s)

   1       19.858498          4.456287
   2        1.135214          1.065464 (ground oscillation)
   3        5.525477          2.350633
   4       29.036367          5.388540
   5        8.333333          2.886751
```

5.4.7 Extended symmetrical eigenvalue problems

Many interesting problems in the field of technical physics cannot be represented as an eigenvalue problem of the type (5.5), but rather as a homogeneous linear system of equation of the type:

$$(A - \lambda S)\mathbf{x} = 0 \tag{5.40}$$

where A is a symmetrical matrix and S is also a symmetrical matrix, which is also *positive definite* (in some physical applications S is called a *structure matrix*).

How can we now reduce the *extended eigenvalue problem* (5.40) to the form

$$(A' - \lambda I)\mathbf{x} = 0?$$

It is apparently very easy to answer this question. If we multiply eq. (5.40) from left by the inverse of S, we obtain

$$\left(S^{-1}A - \lambda I\right)\mathbf{x} = 0$$

with the new matrix of coefficients $A' = S^{-1}A$.

Unfortunately, this simple way to treat the problem is not very useful in practice, because the new matrix of coefficients has lost the important property of being symmetric, i.e.:

$$A'_{i,j} = \sum_{k=1}^{n} s^{-1}_{i,k}\, a_{k,j}$$

is in general not identical to:

$$A'_{j,i} = \sum_{k=1}^{n} s^{-1}_{j,k}\, a_{k,i}\,.$$

A better, albeit more complicated, method to reformulate the problem is the following:

The starting point is the so-called *Cholesky decomposition* of the symmetrical, definite positive matrix S:

$$S = L\,L^T \tag{5.41}$$

Please note that this decomposition is very similar the *LU decomposition* described in chapter 2 (2.4). The difference is that in the present case a symmetrical, definite positive matrix is represented as a product between a lower triangular matrix L and the upper triangular matrix L^T, which is its transpose about the main diagonal.

If we carry out the Cholesky decomposition, the rest of the procedure is relatively straightforward: inserting Eq. (5.41) in the extended eigenvalue equation (5.40) gives

$$\left(A - \lambda L L^T\right)\mathbf{x} = 0\,.$$

If we now take the matrix L^T out of the parenthesis (from right), we have:

$$\left(A(L^T)^{-1} - \lambda L\right)\left(L^T\,\mathbf{x}\right) = 0$$

and

$$\left(\underbrace{L^{-1}A(L^T)^{-1}}_{C} - \lambda I\right)\left(L^T\mathbf{x}\right) = 0$$

i.e.[5]

$$(C - \lambda E)\mathbf{y} = 0 \qquad \text{with} \quad C = L^{-1}A\left(L^{-1}\right)^T \qquad \text{and} \quad \mathbf{y} = L^T\mathbf{x}\,. \tag{5.42}$$

The eigenvalues of the matrix C are identical to the eigenvalues of A in Eq. (5.40), and it is also easy to show that this matrix is symmetrical.

From the eigenvectors $\mathbf{y}$ of the system (5.42) it is possible to determine the eigenvectors $\mathbf{x}$ of Eq. (5.40) through

$$\mathbf{x} = \left(L^{-1}\right)^T\,\mathbf{y} \tag{5.43}$$

The formulas which permit to carry out the *Cholesky decomposition* numerically can be easily derived. We start from the representation of Eq.(5.41) by components:

$$s_{i,j} = \sum_{k=1}^{n} \ell_{ik}\ell_{jk} = \sum_{k=1}^{j} \ell_{ik}\ell_{jk}\,.$$

The calculation of s_{ij} is carried out column by column (index j), with the condition $i \le j$, due to the triangular form of L.

[5] Here we employed the identity $\left(L^T\right)^{-1} = \left(L^{-1}\right)^T$.

For the first column ($j = 1$) we immediately obtain

$$s_{11} = \ell_{11}^2 \rightarrow \ell_{11} = \sqrt{s_{11}}$$

and

$$s_{i1} = \ell_{i1}\ell_{11} \rightarrow \ell_{i1} = \frac{s_{i1}}{\ell_{11}} \quad \text{for} \quad i = 2, \ldots, n\,.$$

Similarly we obtain for the second column ($j = 2$)

$$s_{22} = \ell_{21}^2 + \ell_{22}^2 \rightarrow \ell_{22} = \sqrt{s_{22} - \ell_{21}^2}$$

and

$$s_{i2} = \ell i1\ell21 + \ell_{i2}\ell_{22} \rightarrow \ell_{i2} = \frac{s_{i2} - \ell_{i1}\ell_{21}}{\ell_{22}} \quad \text{for} \quad i = 3, \ldots, n\,.$$

From these equations we can arrive, without further work, to the general Cholesky formulas:

$$\ell_{jj} = \sqrt{s_{jj} - \sum_{t=1}^{j-1} \ell_{jt}^2}\,,$$

$$\ell_{i,j} = \frac{s_{ij} - \sum_{t=1}^{j-1} \ell_{it}\ell_{jt}}{\ell_{jj}} \quad \text{for} \quad (i = j+1, \ldots, n) \tag{5.44}$$

Two more remarks on these equations:

- Clearly, problems will occur if the argument of the square root becomes zero or negative during the Cholesky calculation. This can however be excluded a priori if the matrix S is definite positive. In the program, before each calculation of the square root, we have to check whether

  $$\left(s_{jj} - \sum_{t=1}^{j-1} \ell_{jt}^2\right) > 0\,.$$

  If this is not the case, the matrix S is not *definite positive* and the calculation must be interrupted!

- An analysis of the equations (5.44) shows that the s_{ij} values needed to calculate the ℓ_{ij} are used only once in the course of the calculation: this gives the possibility to store the values of the L matrix (column by column) on the memory locations occupied by the S matrix. A similar procedure to save space is used in the program LUDCMP.
  Disadvantage: the original S matrix is lost in this way.

5.4.8 The program CHOLESKY.

The numerical implementation of the equation (5.44) is described in the following structure chart nr. 23 CHOLESKY. To be precise, in the first (smaller) part of this program. The second part of CHOLESKY performs the numerical calculation of the *new* matrix C according to Eq. (5.42). The corresponding part of the algorithm in the structure chart 23 is taken from the Algol program *reduc1* of Martin und Wilkinson[6] I have to confess that so far I had no time to derive this algorithm from Eq.(5.42).

INPUT parameters:

N: Rows and columns of the matrices A and S.

A(,): Components of a real, symmetrical matrix.

S(,): Components of a real, symmtrical,
definite positive matrix.

OUTPUT parameters:

S(,): Components of the tridiagonal matrix L.

A(,): Components of the symmetrical matrix C.

ERROR: ERRORdiagnostics: ERROR=0 no ERROR occurred
ERROR=1 Attention:
Input Matrix S is not definite positive.

[6] Martin and Wilkinson, Num. Math. **11**, 99 (1968); *Handbook for Autom. Computing*, vol. II, p. 303 (1971).

Structure chart 23 — CHOLESKY(N,A,S,ERROR)

```
ERROR:=0
I=1(1)N
    J=I(1)N
        X:=S(I,J)
        K=1(1)I-1
            X:=X-S(I,K)*S(J,K)
        J ≠ I
        Y:  S(J,I):=X/Y
        N:  X ≤ 0.0
            Y:  ERROR:=1
                print 'Matrix S not def.pos.'
                (return)
            N:  ......
            Y:=SQRT(X)
            S(I,I):=Y
Note: End of the Cholesky Decomposition.
Note: Now the calculation of the matrix C starts (Attention: overwritten on A):
I=1(1)N
    Y:=B(I,I)
    J=I(1)N
        X:=A(I,J)
        K=1(1)I-1
            X:=X-S(I,K)*A(J,K)
        A(J,I):=X/Y
J=1(1)N
    I=J(1)N
        X:=A(I,J)
        K=J(1)I-1
            X:=X-A(K,J)*S(I,K)
        K=1(1)J-1
            X:=X-A(J,K)*S(I,K)
        A(I,J):=X/S(I,I)
I=1(1)N-1
    J=I+1(1)N
        A(I,J):=A(J,I)
        S(I,J):=0.0
```

TEST EXAMPLE:
=============

Numerical Solution of the extended eigenvalue problem:

```
       [A - lambda . B] x  =  0
```

Matrix A (symmetrical)

```
 5.0000   4.0000   1.0000   1.0000
 4.0000   5.0000   1.0000   1.0000
 1.0000   1.0000   4.0000   2.0000
 1.0000   1.0000   2.0000   4.0000
```

Matrix B (symmetrical and definite positive):

```
 5.0000   7.0000   6.0000   5.0000
 7.0000  10.0000   8.0000   7.0000
 6.0000   8.0000  10.0000   9.0000
 5.0000   7.0000   9.0000  10.0000
```

The program CHOLESKY extracts from the original matrices
A and B through the Cholesky-decomposition of B:

```
               B = L . L^T
```

the matrix

```
C = L^(-1) . A . (L^(-1))^T
```

The lower triangular matrix L is:

```
   2.2361      0.0000      0.0000      0.0000
   3.1305      0.4472      0.0000      0.0000
   2.6833     -0.8944      1.4142      0.0000
   2.2361     -0.0000      2.1213      0.7071
```

This marix C is symmetrical and has the same
eigenvalues as the original extended eigenvalue problem:

```
   1.0000     -3.0000     -3.4785      7.9057
  -3.0000     18.0000     16.4438    -41.1096
  -3.4785     16.4438     18.0000    -43.0000
   7.9057    -41.1096    -43.0000    110.0000
```

At this point, using the Jacobi prorgram it is possible to determine
the 4 eigenvalues of the matrix C.

These are: 0.2623 2.3078 1.1530 143.2769

5.4.9 'More advanced programs'

The Jacobi method works in principle extremely well for all real symmetrical matrices. Furthermore most professional libraries offer even more performant programs which are clearly faster for matrices of higher order (> 20)[7].
The basic idea of these programs is a *two-step method*:

1. The symmetrical matrix A is reduced to an easier form, using a *final* (i.e. non-iterative) method. This is typically a *tridiagonal symmetrical* form. The *Householder* algorithm is often used for this.

2. The eigenvectors and eigenvalues of A are calculated from this tridiagonal matrix using the so-called *QR*- or *QL*- algorithms.

The Householder, QR and QL methods cannot be discussed in detail in this lecture. A very good and compact description of the theory and of the corresponding programs RED2 (Householder) und TQLI (QL-Algorithmus) can be found in in [9] and [10].

5.5 Eigenvalues of generic real matrices

This problem is treated in numerical analysis in the same way as that of symmetrical matrices, i.e. with a two-step method. First the matrix A is recast in an easier form through a non-iterative algorithm (see below), and then the eigenvalues of this matrix are calculated.

The basic idea is based on the following statement from matrix theory:
Every matrix can be reduced into a upper Hessnberg form (UHF) using a non-iterative similarity transformation.

and

In an upper-Hessenberg matrix all the components of the matrix below the first lower diagonal are zero.

5.5.1 Transformation of a matrix into an upper-Hessenberg form.

The transformation of a general real matrix into an UHF employs an algorithm which is similar to the gaussian elimination (see chapter 2, LU decomposition), but with the important difference that in this case each transformation of the matrix must be a similarity operation. In the following we will briefly explain the algorithm:

Let us assume that we want to reduce a 7x7 matrix into the UHF, and that the algorithm has already performed part of the necessary work, reducing

[7] An interesting detail: for a long time, the renowned library LAPACK did not include any Jacobi routine, until the following paragraph appeared in the *User's Guide* of 1995, P. 18: *In the future LAPACK will include routines based on the Jacobi algorithm ..., which are slower than the above routines (QR etc) but can be significantly more accurate.*

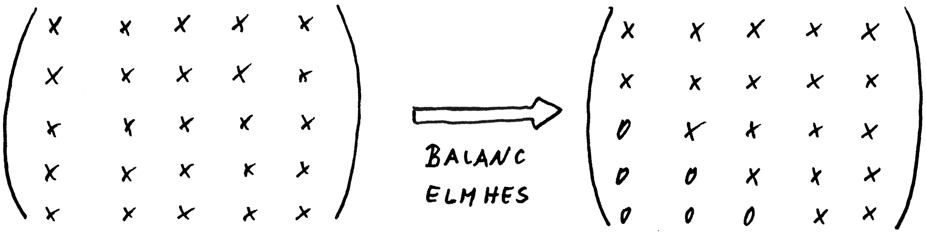


Figure 5.1: Reduction of a general matrix into theupper Hessenberg form.

the first three columns of the matrix into an upper-Hessenberg form. The matrix now looks like this:

```
x  x  x  x  x  x  x

x  x  x  x  x  x  x

0  x  x  x  x  x  x

0  0  x  x  x  x  x

0  0  0  a  x  x  x

0  0  0  b  x  x  x

0  0  0  c  x  x  x
```

In the following we want to reduce the fourth column into UHF, i.e. the last two values of this column must me equal to zero. We can obtain this multiplyinng the third to last row of the matrix by the factors (b/a) and (c/a) and summing this to the last and second before last row.
As you may remember from the LU decomposition, described in chapter 2, in these cases the multiplication factor must be as small as possible: this can be achieved using *pivoting*, i.e. exchanging the relevant rows (in the present case, those which contain a, b and c), so that the largest element between a,b and c occupies the position of 'a'.
There is an important difference between the method we are discussing here and the LU-decomposition. *In order to ensure that the transformation of the matrix A into UHF is a similarity operation, each time an operation is performed on the rows of A, an analogous operation must be performed also on its columns. This applies, in particular, to the exchange of two rows in the pivoting and to the addition of rows to get the Hessenberg zeroes.*

5.5.2 The program ELMHES.

The program ELMHES reduces a generic real matrix into a the corresponding 'Upper-Hessenberg-Matrix'.

Source: [9], P. 752; [10], P. 485f.

INPUT parameters:

A(,): Components of a real matrix.

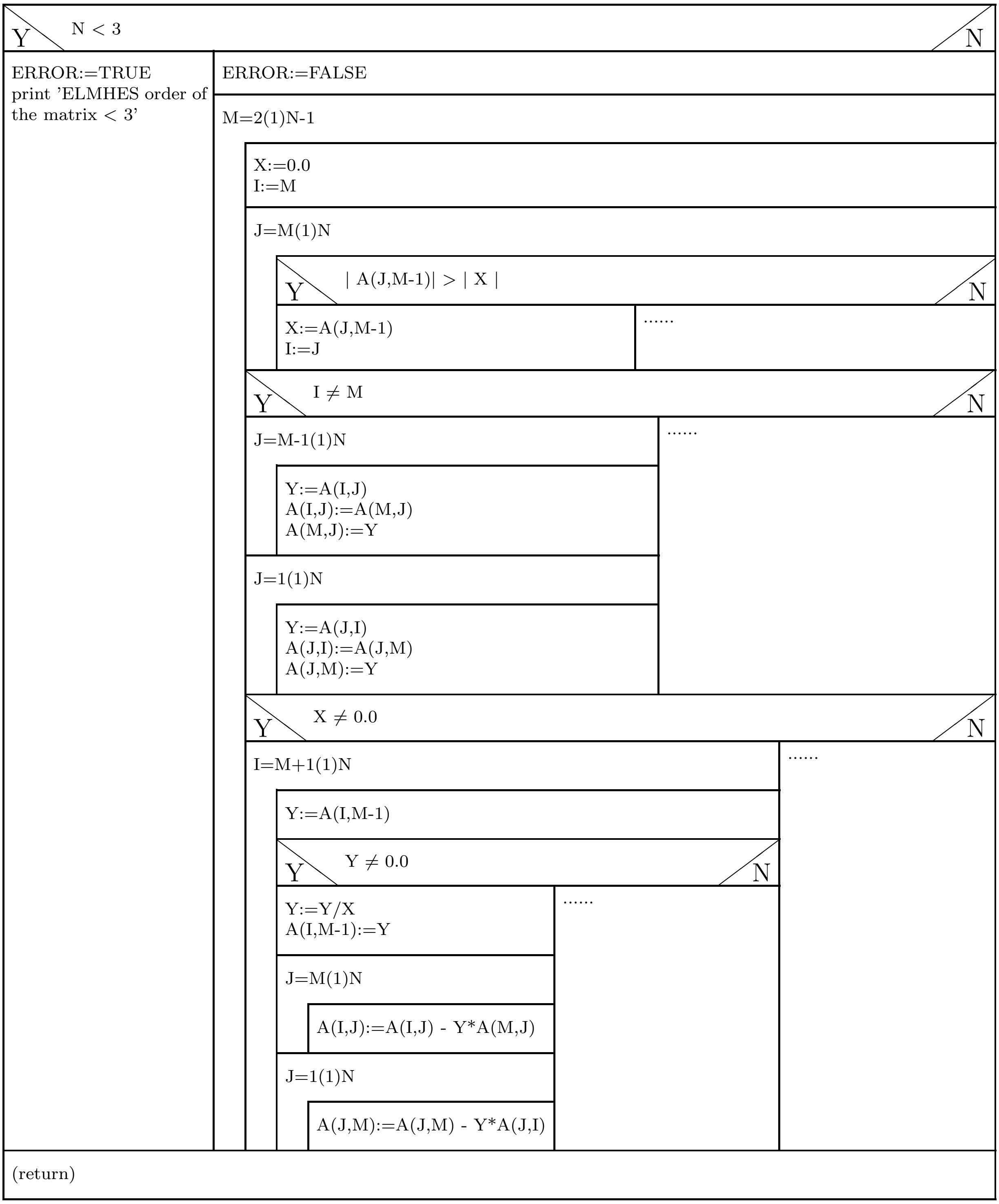
N: Number of rows and columns of the matrix.

OUTPUT parameters:

A(,): 'Upper-Hessenberg-Matrix' with the same eigenvalues as the original matrix.

ERROR: TRUE if the order of the matrix is smaller than 3, FALSE otherwise.

Struktogramm 24 — ELMHES(A,N,ERROR)



```
N < 3
Y
    ERROR:=TRUE
    print ’ELMHES order of the matrix < 3’
N
    ERROR:=FALSE
    M=2(1)N-1
        X:=0.0
        I:=M
        J=M(1)N
            | A(J,M-1)| > | X |
            Y
                X:=A(J,M-1)
                I:=J
            N
                ......
        I ≠ M
        Y
            J=M-1(1)N
                Y:=A(I,J)
                A(I,J):=A(M,J)
                A(M,J):=Y
            J=1(1)N
                Y:=A(J,I)
                A(J,I):=A(J,M)
                A(J,M):=Y
        N
            ......
        X ≠ 0.0
        Y
            I=M+1(1)N
                Y:=A(I,M-1)
                Y ≠ 0.0
                Y
                    Y:=Y/X
                    A(I,M-1):=Y
                    J=M(1)N
                        A(I,J):=A(I,J) - Y*A(M,J)
                    J=1(1)N
                        A(J,M):=A(J,M) - Y*A(J,I)
                N
                    ......
        N
            ......
(return)
```

5.5.3 Determination of the eigenvalues of a matrix in UHF.

In order to calculate the real and complex-conjugate eigenvalues of a real Upper-Hessenberg Matrix we can employ the QR algorithm of Sect. 6.4.9. An example of such a program can be found in Numerical Recipes (Theory and FORTRAN program in [9], P. 374ff; PASCAL programm i [9], theory and C Program in [10], P. 486ff.

If one just wants to obtain the *real* eigenvalues and eigenvectors of a matrix in UHF there is also a method which is much easier than the QR.

As shown in Sect. 6.1 the eigenvalues of a matrix coincide with the zeroes of its *characteristic polynomial*

$$P_n(\lambda) = \lambda^n + \sum_{i=1}^{n} p_i\, \lambda^{n-i}\,.$$

There are several algorithms which permit to calculate the coefficients p_i of such a polynomial. Once the p_i are known, the zeroes of the polynomial can be found using a program for the search of zeroes. This method is however very cumbersome and also prone to roundoff ERRORs and therefore it's little used in practice.
In case of an upper Hessenberg matrix one can however estimate the characteristic polynomial *without knowing explicitely its coefficients*; this will be explained in the next section.

5.5.4 The method of Hyman.

Hyman was able to prove that for Hessenberg matrices the polynomial $P_n(\lambda)$ has the form:

$$P_n(\lambda) = (-1)^{n+1} \cdot a_{21} a_{32} \cdots a_{n,n-1} \cdot H(\lambda).$$

Obviously the zeroes of the polynomial coincide with the zeroes of the function $H(\lambda)$. The values of the function $H(\lambda)$ are given by the following simple linear system of equations, which is also well conditioned numerically:

$$(A - \lambda \cdot I) \begin{pmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ \cdot \\ x_{n-1} \\ 1 \end{pmatrix} = H(\lambda) \begin{pmatrix} 1 \\ 0 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ 0 \end{pmatrix} \quad . \tag{5.45}$$

Here A is the given Hessenberg matrix and $(x_1, x_2, \ldots, x_{n-1}, 1)$ is an auxiliary vector.

Simple algebra shows that the $n-1$ unknown coefficients of the vector $\mathbf{x}$ can be estimated from the following relations:

$$x_n = 1$$

$$x_i = \frac{1}{a_{i+1,i}} \left[\lambda\, x_{i+1} - \sum_{l=0}^{n-i-1} a_{i+1,n-l} x_{n-l} \right] \qquad i = n-1, n-2, \ldots, 2, 1. \tag{5.46}$$

From the first equation of (5.45) it follows:

$$H(\lambda) = \sum_{i=1}^{n} a_{1,i}x_i - \lambda x_1 \quad . \tag{5.47}$$

In this way the calculation of the functional values of $H(\lambda)$ can be very efficiently combined with an efficient program for the search of zeroes (for example INTSCH, see Chapter 5).

The function HYMAN

INPUT PARAMETERS

UHM: Real upper Hessenberg matrix

N: Rows and columns of the matrix

LAM: Argument of the function (λ)

Structure chart 25 — FUNCTION HYMAN(UHM,N,LAM)

```
X(N):=1.0
I=N-1(-1)1
    SUM:= LAM*X(I+1)
    L=0(1)N-I-1
        SUM:=SUM - UHM(I+1,N-L)*X(N-L)
    X(I):= SUM/UHM(I+1,I)
SUM:=-LAM*X(1)
I=1(1)N
    SUM:=SUM + UHM(1,I)*X(I)
HYMAN:=SUM
(return)
```

A test example

Reducing the original matrix in UHF through ELMHES, and using this matrix as an input for HYMAN, we obtain the values of of the function $H(\lambda)$. The zeroes of this curve are the (real) eigenvalues of the matrix. In order to find these zeroes numerically we have to combine the programs ELMHES, HYMAN and INTSCH appropriately.

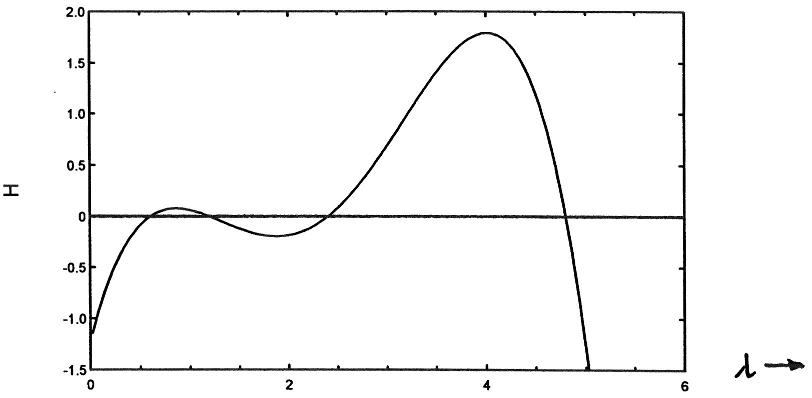


Figure 5.2: The Hyman function $H(\lambda)$ for the 4x4-Matrix in page 207.

Fig. 5.2 shows the function $H(\lambda)$ for the 4x4 matrix of page 207. The four eigenvalues are: 0.6, 1.2, 2.4 and 4.8.

5.5.5 Eigenvalues of tridiagonal matrices.

The Hyman's method can be particularly easily applied to tridiagonal matrices of the form

$$\begin{pmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & c_3 & \\ & & \ddots & \ddots & \ddots \\ & & & \ddots & c_{n-1} \\ & & & a_n & b_n \end{pmatrix}$$

i.e. to matrices which can be specified through three diagonal vectors **a**, **b** and **c**. We obtain through (5.46) and (5.47) the following relations

$$\begin{aligned} x_n &= 1 \\ x_{n-1} &= -\frac{1}{a_n}(b_n - \lambda) \\ x_{n-2} &= -\frac{1}{a_{n-1}}\left(c_{n-1} - \lambda x_{n-1} + b_{n-1}x_{n-1}\right) \\ x_{i-1} &= -\frac{1}{a_i}\left((b_i - \lambda)x_i + c_i x_{i+1}\right) \qquad i = n-2, n-1, \ldots, 3, 2 \\ H(\lambda) &= (b_1 - \lambda)\, x_1 + c_1 x_2 \,. \end{aligned} \tag{5.48}$$

We now show an example (in C) for the numerical calculation of the real eigenvalues of a general, real tridiagonal matrix:

Program Structure in C

```
.
#include <stdio.h>
#include <math.h>
#include "nrutil.c"

int n;
double *a,*b,*c;


double hymtri(double lam)
// This routine permits to calculate the 'Hyman function'
// for tridiagonal matrices through eq. (6.43).
// The size of the matrix (n) and the vectors a, b and c
// are defined globally
{
  int i;
  double x1,x2,x;

  if(n <= 2) {
    printf("Order too small\n");
    return 0.0;
  }

  else {
    x2=-(b[n]-lam)/a[n];
    x1=-((b[n-1]-lam)*x2 + c[n-1])/a[n-1];
    for(i=n-2;i>1;i--){
      x= -((b[i]-lam)*x1 + c[i]*x2)/a[i];
      x2=x1;
      x1=x;
    }
    return (b[1]-lam)*x1 + c[1]*x2;
  }
}


#include "intsch.c"






/********************** main program ******************/
void main()
{
  int nmax=100;
  double *zeroes;
```

```
   .
   .
  n=....;       //Size of the matrix

  a=dvector(1,n);
  b=dvector(1,n);
  c=dvector(1,n);
  zeroes=dvector(1,n);

// The vectors a b c of the tridiagonal matrix are read in:

   .
   .

// Eingabe der INTSCH-Parameter:
  ainit=....;     // Begin of the gross search interval
  aend=....;    // End of the gross search interval
  h=....;       // Stepsize in the gross search interval
  gen=....;     // rel. precision of the eigenvalues


  intsch(&hymtri,ainit,aend,h,gen,nmax,zeroes,&n);

// Output of the zeroes = Eigenvalues of the tridiagonal matrix
   .
   .

  free_dvector(a,1,n);
  free_dvector(b,1,n);
  free_dvector(c,1,n);
  free_dvector(zeroes,1,n);
}
```

Remark: The program for the search of zero INTSCH is explained in detail in section 4.5.2. The call to INTSCH in the above program contains one extra parameter, i.e. the actual name of the function whose zeroes have to be found by INTSCH. In this specific case this is the 'Hyman function' for tridiagonal matrices, called 'hymtri'.
The possibility of passing the names of the routines as parameters is offered by many programming languages (C, F90, Matlab...).

For example, if you use C, you should write the headline of INTSCH as follows:

```
void intsch(double (*fct)(double),double ainit, double aend, double h,
            double gen, int nmax, double zeroes[], int *n)
```

Furthermore, every time you call the function in INTSCH you have to do it as follows:

```
instead of  ... fct(x) ...         ... (*fct)(x) ...
```

Chapter 6

Numerical methods for ordinary differential equations: initial value problems.

6.1 General considerations.

Finding the roots of differential equations is one of the most important problems of numerical mathematics. There are two main reasons:

- Although there is a large variety of analytical methods for the solution of differential equations, it is often so difficult to find the solution that this route is not convenient. With this, we wish by no means to downplay the importance of analytical methods. In fact, in this lecture we follow the motto of the numerics book of Dorn/McCracken [7] *'Numerical methods are no excuse for poor analysis'*.
- Many differential equations do not admit a solution that can be represented in a closed analytical form. This is for example true for the seemingly 'innocent' differential equation:

  $$y'(x) = x^2 + y(x)^2!$$

  However, in these cases a numerical solution is still possible.

In this chapter we will consider *explicit* differential equations, i.e. those that can be solved for the highest possible derivative. We will therefore not treat differential equations of the type:

$$y'(x) + \log y'(x) = 1$$

Furthermore, the methods presented in the following are limited to *first order* differential equations (or systems thereof). However, this limitation is not severe, since higher-order explicit differential equations can always be recast into a system of first order differential equations.
In fact, the differential equation of n^{th} order

$$y^{(n)} = F(x; y, y', y'', \dots, y^{(n-1)})$$

is equivalent to the system:

$$\begin{array}{ll}
y_1' = y_2 & \equiv f_1(x) \\
y_2' = y_3 & \equiv f_2(x) \\
. & \\
. & \\
. & \\
y_{n-1}' = y_n & \equiv f_{n-1}(x) \\
y_n' = F(x; y_1, y_2, \ldots, y_n) & \equiv f_n(x)
\end{array}$$

If we rewrite y_i', y_i und f_i as components of a vector this system can be rewritten in the form

$$\mathbf{y}'(x) = \mathbf{f}(x; \mathbf{y}). \tag{6.1}$$

Numerical methods for approximate solution of differential equations obviously do not return a function with all the integration constants, but calculate point by point the solutions specified by appropriate *boundary conditions*. In this chapter we will treat only *initial value problems*, i.e. those for which the boundary conditions are specified by the behaviour of the solution $\mathbf{y}(x)$ in a point $x = x_o$. A *fully specified initial value problem* has therefore the form

$$\mathbf{y}'(x) = \mathbf{f}(x; \mathbf{y}), \qquad \text{with} \quad \mathbf{y}(x_o) = \mathbf{y}_o \quad . \tag{6.2}$$

6.2 Expansion of the solution in Taylor series.

The starting point for all further considerations is the expansion of the i-th solution of the system (6.2) in a power series:

$$y_i(x) = \sum_{\nu=0}^{p} \frac{(x - x_o)^\nu}{\nu!} \left[\frac{d^\nu}{dx^\nu} y_i(x) \right]_{x_o, \mathbf{y}_o} + R_i \tag{6.3}$$

with the *Lagrange remainder*

$$R_i = \frac{(x - x_o)^{p+1}}{(p+1)!} \left[\frac{d^{p+1}}{dx^{p+1}} y_i(x) \right]_{x=\xi} \qquad x_o \le \xi \le x \quad . \tag{6.4}$$

The aim of all the following methods is to find an approximate value for the y_i's in $x_o + h$, starting from a starting point x_0. p is called the *order of the method*:

$$\hat{y}_i(x_o + h) = \sum_{\nu=0}^{p} \frac{h^\nu}{\nu!} \left[\frac{d^\nu}{dx^\nu} y_i(x) \right]_{x_o, \mathbf{y}_o} \quad . \tag{6.5}$$

Remark: Here and in the following $\hat{y}$ indicates the approximate value of y.

6.3 Euler's method.

The oldest approximate method to numerically solve differential equations goes under the name of Leonhard Euler. In this *first order* method (p=1) the expansion (6.5) is truncated right after the second term, yielding:

$$\hat{y}_i(x_o + h) = y_i(x_o) + h \cdot y'(x) \mid_{x=x_o} = y_i(x_o) + h \cdot f_i(x_o, \mathbf{y}_o) \tag{6.6}$$

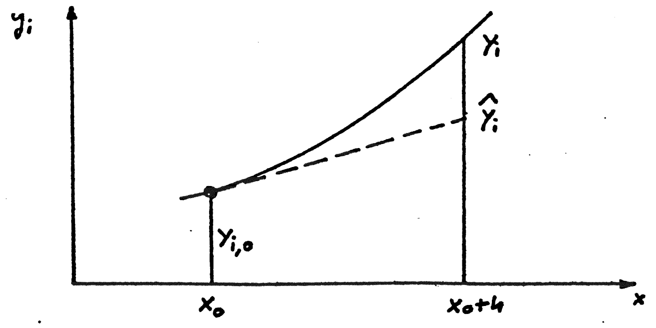


Figure 6.1: Geometrical interpretation of Euler's method.

The geometrical interpretation of this formula is quite simple: the exact solution $y_i(x)$ is approximated by its tangent in x_o (see Fig. 6.1).
It is clear that Euler's method can give meaningful results only for very small values of the step size (h).

6.4 Runge-Kutta Methods.

Runge-Kutta methods are among the most popular numerical methods for initial value problems. In fact, they are well suited for computers and in many cases can give results with high precision at an acceptable computational cost. The biggest disadvantage - as we will show - of the Runge-Kutta methods is that it is quite difficult to make a reliable error estimate (see section 6.4.5).

Runge-Kutta methods are characterised by the three following properties:

- They derive from a Taylor series expansion of the solution, truncated after the h^p term. p is called the *order* of the Runge-Kutta method.
- They belong to the class of *one-step methods*, i.e. in order to calculate the function in the point $x + h$ it is enough to know the information in the previous point x.
- In order to calculate the approximate value of $y_i(x)$ one needs to know only the value of the functions $f_i(x, \mathbf{y})$, *but not their derivatives!* This is a big practical advantage over Taylor series expansions.

From these three characteristics of the Runge-Kutta methods it follows:

Euler's method is a Runge-Kutta method of first order.

6.4.1 Second-order Runge-Kutta methods.

We will now show how to derive second order Runge-Kutta formulas. For the sake of simplicity, we will consider a 'differential system' which consists of the single equation

$$y'(x) = f(x, y)$$

with the initial condition

$$y(x_o) = y_o.$$

The starting point for a Runge-Kutta formula of second order is the *ansatz*

$$\hat{y}(x_o + h) = y_o + h \cdot (c_1 g_1 + c_2 g_2) \quad . \tag{6.7}$$

In (6.7) g_1 and g_2 represent values of the function $f(x, y)$, and in particular

$$\begin{array}{rcl} g_1 & = & f(x_o, y_o) \\ g_2 & = & f(x_o + a_2 h, y_o + b_{2,1} h g_1) \end{array} \tag{6.8}$$

c_1, c_2, a_2 and $b_{2,1}$ are the *Runge-Kutta coefficients.* These coefficients have to be chosen so that the *ansatz* above coincides (approximately) with the Taylor series expansion for $p = 2$.

$$\hat{y}(x_o + h) = y_o + h \cdot f(x_o, y_o) + \frac{h^2}{2} \left[f_x + f \cdot f_y \right]_{x_o, y_o} \tag{6.9}$$

In order to satisfy this relation, we expand g_2 (6.8) in $h = 0$ in powers of h and truncate the expansion after the linear term:

$$\begin{array}{rcl} g_2(h) & = & g_2(h = 0) + h \cdot \left[\frac{d}{dh} g_2(h) \right]_{h=0} \qquad (6.10) \\ & = & f(x_o, y_o) + h \cdot [f_x(x_o, y_o) \cdot a_2 + f(x_o, y_o) \cdot f_y(x_o, y_o) \cdot b_{2,1}] \end{array}$$

If we now insert (6.10) with $g_1 = f(x_o, y_o)$ in the Runge-Kutta *ansatz* (6.7), we obtain:

$$\begin{array}{rcl} \hat{y}(x_o + h) & = & y_o + h \cdot c_1 \cdot f(x_o, y_o) + h \cdot c_2 \cdot f(x_o, y_o) + \qquad (6.11) \\ & & + h^2 \cdot c_2 \cdot a_2 \cdot f_x(x_o, y_o) + h^2 \cdot c_2 \cdot b_{2,1} \cdot f(x_o, y_o) \cdot f_y(x_o, y_0) \quad . \end{array}$$

Setting (6.9) equal to (6.11) and equating the coefficients, we obtain the following *non-linear set of equations for the Runge-Kutta coefficients of second order*:

$$\begin{array}{rcl} c_1 + c_2 & = & 1 \\ c_2 \cdot a_2 & = & \frac{1}{2} \\ c_2 \cdot b_{2,1} & = & \frac{1}{2} \end{array} \tag{6.12}$$

Notice that this system is *underdetermined*: we have only three equations for four coefficients! *One of the four coefficients is not fixed, and can be chosen arbitrarily.* As a consequence, there is no <u>unique</u> definition of a second-order Runge-Kutta method. Indeed, there are <u>infinitely many</u> possible choices for the coefficients, and it is left only to the ability of the user to choose one of them, so that the remaining three values are as simple as possible.

<u>Two examples:</u>

- With the choice $c_1 = 0$ one obtains for

$$c_2 = 1 \qquad a_2 = \frac{1}{2} \qquad \text{und} \qquad b_{2,1} = \frac{1}{2} \quad .$$

The Runge-Kutta formula which results from (6.7)

$$\hat{y}(x_o + h) = y_o + h \cdot f(x_o + \frac{h}{2}, y_o + \frac{h}{2} f(x_o, y_o)) \tag{6.13}$$

is called the *modified Euler method.*

- With the choice $c_1 = 1/2$ one obtains

$$c_2 = \frac{1}{2} \qquad a_2 = 1 \qquad b_{2,1} = 1 \quad .$$

The corresponding solution

$$\hat{y}(x_o + h) = y_o + h \left\{ \frac{1}{2} f(x_o, y_o) + \frac{1}{2} f\left[x_o + h, y_o + hf(x_o, y_o)\right] \right\} \tag{6.14}$$

is called the *improved Euler's method.*

A graphical interpretation of these two approximate formulas is sketched in Fig.6.2.

- Modified Euler's formula: The solution is approximated by a straight line through (x_o, y_o) with the slope of $y(x)$ in the middle point of the interval $[x_o, x_o + h]$.
- Improved Euler's formula. The solution is approximated by a straight line through (x_o, y_o), whose slope is the arithmetical average between $y'(x_o)$ und $\hat{y}'(x_o + h)$.

In Figure 6.3 the Runge-Kutta formulas described so far – (6.6), (6.13) and (6.14) – are applied to the simple example

$$y'(x) = e^x \qquad y(0) = 0.$$

The exact solution is

$$y(x) = e^x - 1 \quad ,$$

and we wish to find $\hat{y}(1)$.
From Fig. 6.3 it is clear that the two second order Runge-Kutta methods are much better than the first order Euler's method.
However, it would be wrong to conclude that the 'modified Euler's formula' is a better approximation than the 'improved Euler's formula', based on this particular example. There are in fact other examples in which the situation is exactly the opposite!

All Runge-Kutta formulas of the same order are in principle equally good!

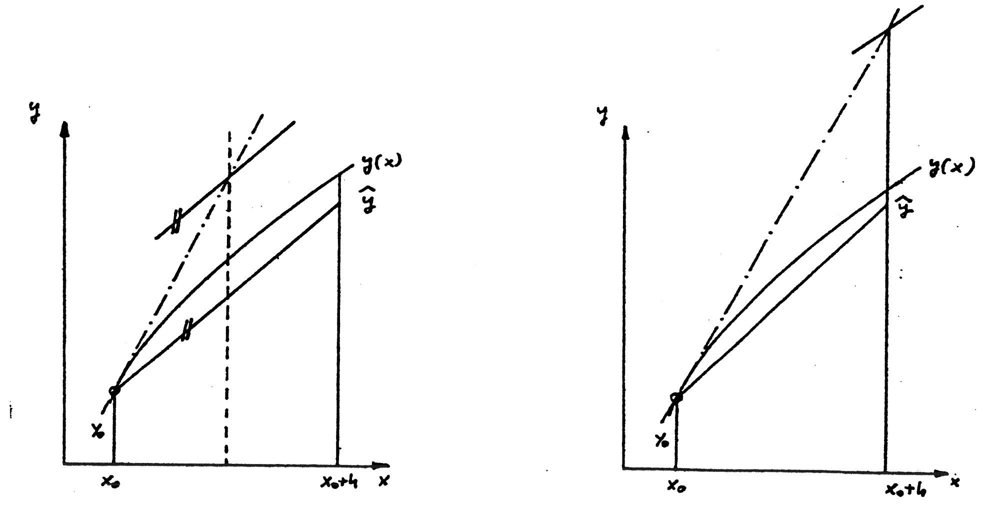


Figure 6.2: Graphical interpretation of the 'modified Euler's formula' (left) and of the 'improved Euler's formula' (right).

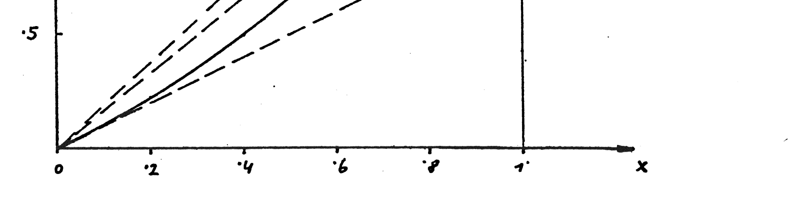


Figure 6.3: A test for first and second order Runge-Kutta methods.

6.4.2 Higher-order Runge-Kutta methods.

For differential systems of n equations the *general Runge-Kutta ansatz of p-th order* holds:

$$\hat{y}_i(x_o + h) = y_{i,o} + h \cdot \sum_{j=1}^{p} c_j g_{i,j} \tag{6.15}$$

with

$$g_{i,1} = f_i(x_o; y_{1,o}, y_{2,o}, \ldots, y_{n,o}) \tag{6.16}$$

and

$$\begin{aligned} g_{i,j} \quad &= \quad f_i(x_o + a_j h; y_{1,o} + h \sum_{\ell=1}^{j-1} b_{j,\ell} g_{1,\ell}, \ldots, \\ & \qquad y_{n,o} + h \sum_{\ell=1}^{j-1} b_{j,\ell} g_{n,\ell}) \end{aligned} \tag{6.17}$$

with $i = 1, \ldots, n$ and $j = 1, \ldots, p$.
[(6.17) ist clearly a *recursion formula*: For the calculation of $g_{i,j}$ we need to know all previous g-values $g_{k,\ell}$, $\ell = 1, \ldots, j - 1$ and $k = 1, \ldots, n$.]

The Runge-Kutta ansatz (6.15) requires exactly $(p^2 + 3p - 2)/2$ coefficients, i.e.

- the p coefficients $\quad c_1, \ldots, c_p \quad$,
- the $p - 1$ coefficients $\quad a_2, \ldots, a_p \quad$ and
- the $p(p - 1)/2$ coefficients $\quad b_{2,1}, \ldots, b_{p,p-1} \quad$.

For improved clarity these numbers are usually represented in a diagram of this form:

a_2	$b_{2,1}$				
a_3	$b_{3,1}$	$b_{3,2}$			
$\vdots$	$\vdots$	$\vdots$	$\ddots$		
a_p	$b_{p,1}$	$b_{p,2}$	$\ldots$	$b_{p,p-1}$	
	c_1	c_2	$\ldots$	c_{p-1}	c_p

The calculation of all coefficients works in principle exacly as illustrated for the case $p = 2$. If p is large, however, this can be quite demanding (see for example [19]). Also for $p > 2$, one finds an underdetermined set of equations for the coefficients, which leads to an infinite manifold of Runge-Kutta formulas.

Obviously, for practical uses one does not need to derive all coefficients from scratch, but can resort to the many ready-to-use formulas available in literature. A collection of Runge-Kutta formulas of order 1-8 can be found, for example, in [2], P. 215-217.

Nowadays, the two formulas which are most used in practice are fourth-order Runge-Kutta methods, namely

`the 3/8 formula:`

1/3	1/3			
2/3	-1/3	1		
1	1	-1	1	
	1/8	3/8	3/8	1/8

`the 'classical' Runge-Kutta-formula:`

1/2	1/2			
1/2	0	1/2		
1	0	0	1	
	1/6	1/3	1/3	1/6

The program presented in section 6.5 employs the *classical Runge-Kutta formula*. The explicit expression of the solution reads in this case:

$$\hat{y}_i(x_o + h) = y_{i,o} + h\left[\frac{1}{6}g_{i,1} + \frac{1}{3}g_{i,2} + \frac{1}{3}g_{i,3} + \frac{1}{6}g_{i,4}\right] \tag{6.18}$$

with

$$\begin{aligned} g_{i,1} &= f_i(x_o; y_{1,o}, \ldots, y_{n,o}) \\ g_{i,2} &= f_i(x_o + \frac{h}{2}; y_{1,o} + \frac{h}{2}g_{1,1}, \ldots, y_{n,o} + \frac{h}{2}g_{n,1}) \\ g_{i,3} &= f_i(x_o + \frac{h}{2}; y_{1,o} + \frac{h}{2}g_{1,2}, \ldots, y_{n,o} + \frac{h}{2}g_{n,2}) \\ g_{i,4} &= f_i(x_o + h; y_{1,o} + hg_{1,3}, \ldots, y_{n,o} + hg_{n,3}) \end{aligned} \tag{6.19}$$

and $i = 1, \ldots, n$. As one can see, since the coefficients $b_{3,1}$, $b_{4,1}$ and $b_{4,2}$ are zero, the sums in (6.17) reduce to a *single* term.

Remark: The classical Runge-Kutta method, applied to the example at page 127, would give an error of 0.034%.

6.4.3 Use of Runge-Kutta formulas:

The formulas (6.18) and (6.19) permit to compute the approximate value of a function in $x = x_o + h$, starting from known initial values $y_{i,o}$ in $x = x_o$.

Let us imagine, however, that we wish to calculate the approximate value of the function not only for a single point $x_1 = x_o + h$, but also for other values of the abscissas $\cdots x_4 > x_3 > x_2 > x_1$! This means that we have to apply the Runge-Kutta method not only once, but several times one after each other. We thus want to move away from a starting point x_o "step by step":

$$\hat{y}_i(x_o + h) \equiv \hat{y}_{i,1} = y_{i,o} + h\sum_{j=1}^{p} c_j g_{i,j}(x_o; y_{1,o}, \ldots, y_{n,o})$$

$$\hat{y}_i(x_o + 2h) \equiv \hat{y}_{i,2} = \hat{y}_{i,1} + h\sum_{j=1}^{p} c_j g_{i,j}(x_o + h; \hat{y}_{1,1}, \ldots, \hat{y}_{n,1})$$

There is an obvious difference between the first and all further Runge-Kutta moves. In fact, all x/y values on the right hand side of the first equation are initial values for the problem, and thus are known exactly. In all other equations, the x/y are instead approximate values!

6.4.4 An example: quality problem.

We now assume to have at our disposal a program RUNGETEST that can solve the initial value problem (6.2) using the formulas (6.18) and (6.19). *The stepsize h, decided by the user, is constant in the whole Runge-Kutta process.*

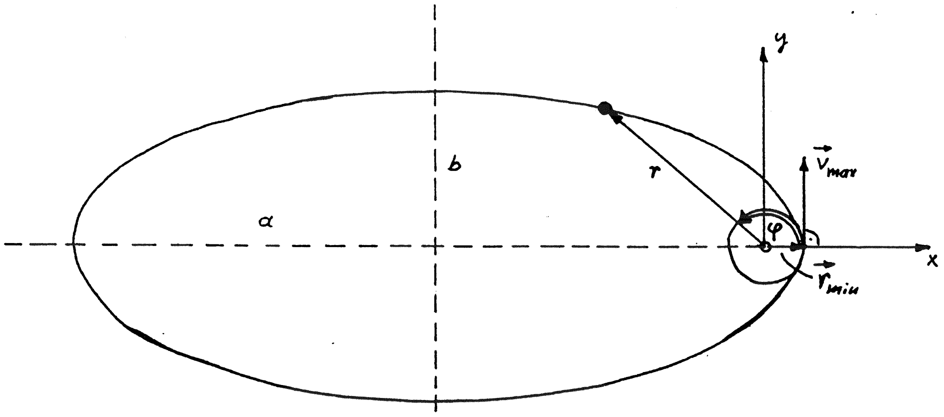


Figure 6.4: Trajectory of an earth satellite.

We wish to assess the performance of RUNGETEST on a non-trivial problem: the calculation of the trajectory of an satellite orbiting around the earth (see Fig.6.4).

Definition of the Problem:

A satellite is set into motion starting from the surface of the earth, with a tangential velocity v_{max}. We wish to calculate its trajectory under the following ideal assumptions:

- $v_{max} <$ escape velocity.
- The earth is an ideal, homogeneous sphere with radius r_{min}.
- The influence of the atmosphere is negligible.
- The influence of other celestial bodies is also negligible.

Under these assumptions the *equation of motion* of the satellite reads

$$\ddot{\mathbf{r}} = -\frac{\gamma M}{r^3}\mathbf{r} \tag{6.20}$$

with $\gamma = 6.67 \cdot 10^{-11}\ m^3/kgs^2$ (gravitational constant) and $M = 5.977 \cdot 10^{24}\ kg$ (mass of the earth). $\mathbf{r}$ is the radius vector connecting the center of the earth with the instantaneous position of the satellite. Equation (6.20) can be rewritten as a set of two second-order differential equations for (1) the distance r and (2) the angle of rotation φ:

$$\begin{aligned} \ddot{r} &= r\dot{\varphi}^2 - \frac{\gamma M}{r^2} \\ \ddot{\varphi} &= -\frac{2\dot{r}\dot{\varphi}}{r} \end{aligned} \tag{6.21}$$

If we now set

$$r \to y_1 \quad \varphi \to y_2 \quad \dot{r} \to y_3 \quad \dot{\varphi} \to y_4 \quad ,$$

we obtain the following system of 4 first-order differential equations:

$$\begin{aligned} \dot{y}_1 &= y_3 & y_1(t=0) &= r_{min} \\ \dot{y}_2 &= y_4 & y_2(t=0) &= 0 \\ \dot{y}_3 &= y_1 y_4^2 - \frac{\gamma M}{y_1^2} & y_3(t=0) &= 0 \\ \dot{y}_4 &= -\frac{2 y_3 y_4}{y_1} & y_4 &= \frac{v_{max}}{r_{min}} = 58.29527 \end{aligned} \qquad (6.22)$$

For the actual test runs we used $r_{min} = 6.37 \cdot 10^6$ m and $v_{max} = 10.4 \cdot 10^3$ m/s.

If we express all lengths in units of r_{min}, all velocities in units of v_{max}, and all times in units of the exact revolution period, which can be calculated exactly, we obtain for the prefactor in eq. (6.21)

$$\alpha \equiv \gamma M = 1966.39\,.$$

A criterion to assess the reliability of the numerical method is obviously the stability of the trajectory, i.e. the elliptical orbit should remain exactly the same revolution after revolution. Deviations from this stability indicates a methodological and/or rounding error in the Runge-Kutta program![1]

It is clear that the methodological error in the program will be smaller the smaller the stepsize in the Runge-Kutta process. This can be clearly seen in Fig. 6.5, where 5 full revolutions are shown each time, for different step sizes – (a) 1/50, (b) 1/60, (c) 1/70 and (d) 1/80 of the revolution period of the satellite, respectively.

The test clearly shows what the biggest effect of the stepsize on the performance of the Runge-Kutta process is: with a stepsize of 1/50 of the revolution period the process is completely unstable (a). Reducing the stepsize merely by a factor 1.6 (d) the trajectory becomes perfectly stable.

We can thus conclude:

- The program RUNGETEST is not usable in practice!
- In order to be practically usable, a Runge-Kutta program must be able to perform an *error diagnostics* i.e. an *automatical stepsize adjustment!* (adaptive stepsize algorithm).

[1] As shown by a careful analysis of the results of the test, in this example rounding errors play only a minor role.

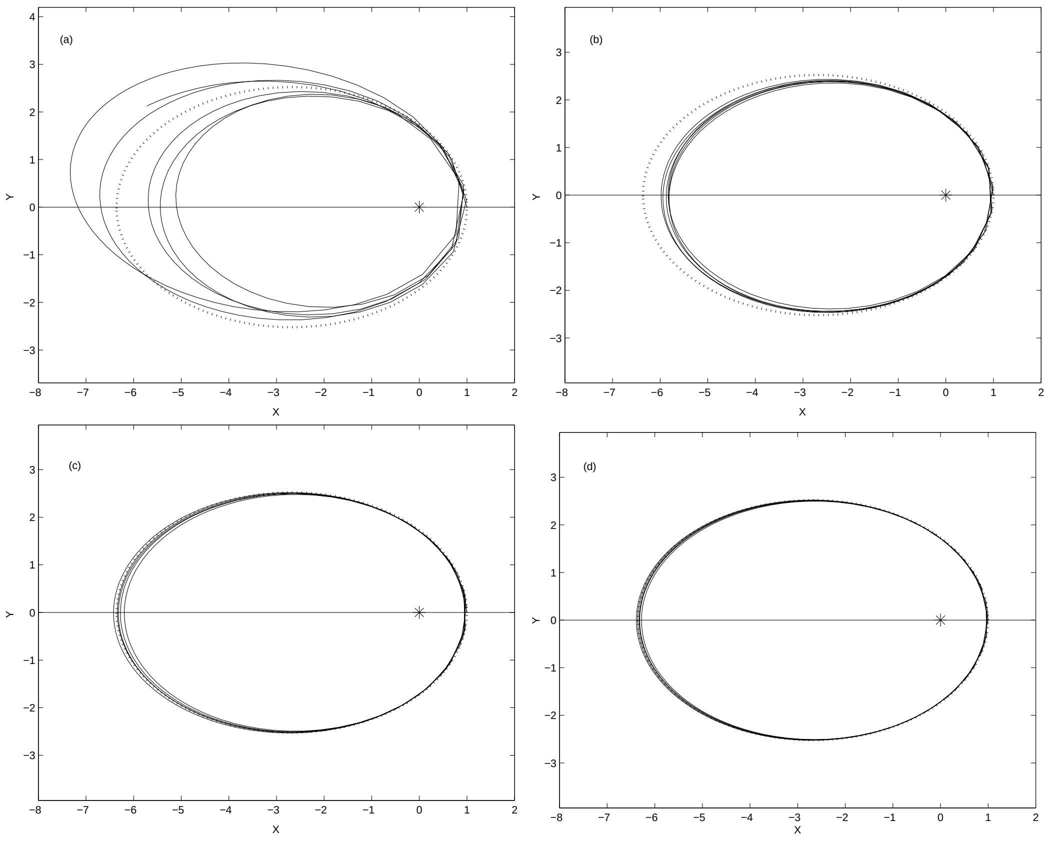


Figure 6.5: Stability test for RUNGETEST. The constant stepsizes have been chosen as follows: (a) $h = 1/50$, (b) $h = 1/60$, (c) $h = 1/70$, (d) $h = 1/80$ of the revolution period. The star indicates the centre of the earth, and the dotted line the exact analytical trajectory of the satellite.

6.4.5 Error estimate and stepsize adaptation in Runge-Kutta methods.

As already discussed, a numerical result can be judged useful or useless, based on its (absolute or relative) error. This is one of the few clues to assess the quality of the approximation.

Since the Runge-Kutta methods are strongly related to the Taylor series expansion (6.3), a natural choice is to use the corresponding Lagrange remainder (6.4) to estimate the methodological error. For the classical Runge-Kutta method (order $p = 4$) this gives:

$$E_V = \frac{h^5}{120}\left[y_i^{(5)}(x)\right]_{x=\xi} \qquad x_o \leq \xi \leq x_o + h$$

and

$$E_V = C(h) \cdot h^5 \quad . \tag{6.23}$$

Since E_V represents the error between the exact value of the solution $y_i(x)$ in the point $x_o + h$ and the corresponding approximate solution $\hat{y}_i(x_o + h)$, we have:

$$y_i(x_o + h) = \hat{y}_i(x_o + h) + C(h) \cdot h^5 \quad . \tag{6.24}$$

It would be equivalent to arrive from x_o to $x_o + h$ with *two successive Runge-Kutta moves of width $h/2$.* This would give:

$$y_i(x_o + h) = \hat{y}_i(x_o + 2 \cdot \frac{h}{2}) + C(h/2) \cdot 2 \cdot \left(\frac{h}{2}\right)^5 \quad . \tag{6.25}$$

In the following, we omit the index i in the y_is, and use the abbreviations:

$$\hat{y}(x_o + h) \equiv \hat{y}(h) \qquad \text{und} \qquad \hat{y}(x_o + 2\frac{h}{2}) \equiv \hat{y}(h/2).$$

We now assume that C does not depend "too strongly" on h, i.e. that to a good approximation:

$$C(h) \approx C(h/2) = C.$$

The constant C and the corresponding methodological error $E_V(h) = Ch^5$ can be calculated setting (6.24) equal to (6.25):

$$E_V(h) = \frac{16}{15}\left[\hat{y}(h/2) - \hat{y}(h)\right] \approx \hat{y}(h/2) - \hat{y}(h) \quad . \tag{6.26}$$

i.e. we can estimate the (absolute) methodological error from the difference between the two Runge-Kutta solutions $\hat{y}(h/2)$ (= approximate solution after two half moves) and $\hat{y}(h)$ (= approximate solution after one full move).

Furthermore, combining equations (6.25) and (6.26) we find the relation that permits *to improve the approximate value $\hat{y}$ in a simple way*:

$$\hat{y}_{improved} = \hat{y}(h/2) + \frac{\hat{y}(h/2) - \hat{y}(h)}{15} \, . \tag{6.27}$$

Hovever, there is a problem connected with this method. Clearly, we obtain a better approximation for the solution, but we have *little information* on the quality of this corrected y! For this reason, this formula is not used in many programs.

We now come to the the final point: we can ask *how large the ideal stepsize h should be, in order to ensure that E_V does not exceed a given error threshold ϵ.*
Since the methodological error is proportional to the fifth power of h, this question is easy to answer. We have in fact:

$$\frac{\epsilon}{E_V(h)} = \left(\frac{h_{ideal}}{h}\right)^5$$

From which we immediately obtain *a definition* for h_{ideal}:

$$h_{ideal} = h \cdot \left(\frac{\hat{y}(h/2) - \hat{y}(h)}{\epsilon}\right)^{-\frac{1}{5}} \tag{6.28}$$

This equation is the basis for an *stepsize adaptation* used for example in program RKQC (sect. 6.5.2). In that program, the actual methodological error is calculated according to (6.26) and compared to the error threshold ϵ. The rest of the program proceeds as follows:

- $\epsilon \geq \hat{y}(h/2) - \hat{y}(h)$: The error of the last Runge-Kutta move is smaller than ϵ. *For the next Runge-Kutta move the stepsize can be increased according to (6.28).*
- $\epsilon < \hat{y}(h/2) - \hat{y}(h)$: The error of the last Runge-Kutta move is larger than ϵ. *The last Runge-Kutta move is repeated with a stepsize is reduced according to (6.28).*

This strategy has two advantages: first, it ensures that the stepsize is always small enough to keep the methodological error below a given threshold ϵ; second, the amplitude of the interval is always as large as possible, which is convenient in terms of computational time.

Finally we would like to recall once more that the method for adaptive stepsize shown here is based on a lot of assumptions (for example, that C is independent on h) which are not always satisfied! Furthemore, so far we have always considered only the *local* error, but we have to keep in mind that in many cases the local errors which occur in different steps can sum up to a much larger *global* error.

The stepsize adaptative algorithm illustrated in the following suites of programs tries to solve some of these problems.

6.5 The programs ODEINT, RKQC and RK4.

Source: [9], P. 550-560, simplified.

The programs ODEINT, RKQC and RK4 permit to numerically solve an initial value problem (6.1).

6.5.1 The program ODEINT.

ODEINT (Ordinary Differential Equations INTegrator) is the 'driver program' of the suite of programs:

INPUT parameters:

YSTART( ): Vector $\mathbf{y}_o$ with the initial values of the system of differential equations.

N: Number n of equations in the system.

X1, X2: Start and end point of the integration interval.

EPS: Required *relative* precision (see the following remarks).

HSTART: Guessed value for the stepsize of the Runge-Kutta process.

HMIN: Minimum value for the stepsize.

NPTMAX: Maximum number of points that can be saved in the arrays XX and YY.

OUTPUT parameters:

NVALUES: Number of stored points.

XX( ): Abscissas of the points.

YY(,): Ordinates of the solution:
first index = index of the function,
second index = label of the abscissas.

Remarks on ODEINT: This program controls the step-by-step functioning of the Runge-Kunta process in the interval X1≤X≤X2. The quality-controlled results produced by the routine RKQC on the sampling points are saved in the arrays XX and YY. One has no influence on the number and distribution of points in the interval [X1,X2].
The approximate $\hat{y}_i$ at the end of the integration interval (X2) are returned to the calling program in the array YSTART. This is practical in case these $\hat{y}_i$-values are needed as initial values for another integration, in an interval immediately after the first.
Another important function of ODEINT is the calculation of the *scaling values* for the error diagnostics. Since EPS indicates a *relative* error threshold, it would meaningful to rescale the moduli of some $\hat{y}_i$-values, i.e. to write

```
YSCAL(I):= |Y(I)| .
```

Such a rescaling would however result in a program interruption in case a zero y_i occurs (Division by zero in RKQC); for this reason the rescaling in ODEINT is performed as follows:

```
YSCAL(I):= |Y(I)| + |H*F(I)| + 'TINY'        ,
```

which avoids this complication.

6.5.2 The program RKQC.

RKQC (Runge-Kutta Quality Control): This program performs the quality control of a single Runge-Kutta move and the corresponding stepsize adaptation.

INPUT parameters:

Y( ): $\hat{y}_i$'s of the last Runge-Kutta move =
initial values for the next step.

F( ): Array of the corresponding f_i-values.

N: Number of the equations of the system.

X: Abscissas of the current initial value.

HTRY: Amplitude of the interval for the next step.

EPS: Required *relative* precision.

YSCAL( ): Scaling factors for the next precision evaluation.

OUTPUT parameters:

Y( ): New $\hat{y}_i$'s.

F( ): Array of the corresponding f_i-values.

X: New abscissas.

HNEXT: Proposed stepsize for the next iteration.

Remarks on RKQC: This program performs the three Runge-Kutta moves needed for the error diagnostics, two with amplitude $h/2$ and one with amplitude h. After this, it gives an approximate estimate of the methodological error for each $\hat{y}_i$, according to (6.26). The maximum value of the *rescaled* methodological error is calculated and saved in ERRMAX. ERRMAX is given by the expression:

$$\mathrm{Max}\,[|\ \hat{y}(h/2) - \hat{y}(h)\ |_{rel}]$$

according to (6.26).

After this a program branching occurs: if

$$\text{ERRMAX} \leq \text{EPS}$$

the Runge-Kutta move was successful. Now, apart from a final correction of $\hat{y}$ according to (6.27), the ideal stepsize for the next step is calculated. The corresponding instruction

$$\text{HNEXT} := \text{SAFETY*H*(ERRMAX/EPS)**}(-1.0/5.0)$$

satisfies eq. (6.28) exactly, up to the "safety constant" SAFETY:=0.9 In case of a very small value of
ERRMAX, the program performs a further safety control[2] In order to avoid that HNEXT grows too fast, in the case

$$\text{ERRMAX/EPS} < \text{ERRCON}$$

the new stepsize is recalculated as follows:

$$\text{HNEXT} := 4. * \text{H}$$

At this point the program jumps back to ODEINT.

In case that

$$\text{ERRMAX} > \text{EPS}$$

the Runge-Kutta move was not successful and therefore must be recalculated with a smaller stepsize. The corresponding instruction

$$\text{H:=SAFETY*H*(ERRMAX/EPS)**}(-1.0/4.0)$$

differs from (6.28) not only for the presence of the "safety constant", but also for the *different exponent*: this should account for the already mentioned fact that the *global* error accumulates much more dramatically than the *local* one. The exponent reduced with respect to (6.28) takes care in this sense of still smaller interval amplitudes and therefore of an even more convenient error behaviour in the Runge-Kutta process.

[2] ERRCON, the 'error control' constant, has the value $6 \cdot 10^{-4}$.

6.5.3 The programs RK4 and DERIVS.

RK4 (Runge-Kutta 4) performs a single Runge-Kutta move according to equations (6.18) and (6.19):

INPUT parameters:

Y( ): Initial value for the current Runge-Kutta move.

G1( ): Array of the f_i values at the current initial point.

N: Number n of equations in the differential system of equations.

X: Abscissa of the initial point.

H: Stepsize for the current Runge-Kutta move.

OUTPUT parameters:

YOUT( ): Runge-Kutta approximate value $\hat{y}_i$ in the point X+H.

DERIVS:

This program, which is called from all three programs in the system, is provided by the user and contains the definition of the functions f_i. In C it would read for example:

```
void derivs(double x, double y[], double f[])
{
  f[1]= .....;  // means f_{1}(x,y1,y2,...)
  f[2]= .....;  // means f_{2}(x,y1,y2,...)
   .
   .
  f[n]= .....;  // means f_{n}(x,y1,y2,...)
}
```

Structure chart 26 — ODEINT(YSTART,N,X1,X2,EPS,HINIT,HMIN,NSTMAX, NVALUES,XX,YY)

```
TINY:=1.0E-30
X:=X1
XX(1):=X
H:=HINIT
I=1(1)N
    Y(I):=YSTART(I)
    YY(I,1):=Y(I)
NSTP=1(1)NSTMAX-1
    DERIVS(X,Y,F)
    I=1(1)N
        YSCAL(I):=|Y(I)| + |F(I)*H| + TINY
    X+H > X2
    Y:
        H:=X2-X
    N:
        ......
    RKQC(Y,F,N,X,H,EPS,YSCAL,HNEXT)
    XX(NSTP+1):=X
    I=1(1)N
        YY(I,NSTP+1):=Y(I)
    X ≥ X2
    Y:
        I=1(1)N
            YSTART(I):=Y(I)
        NVALUES:=NSTP+1
        (return)
    N:
        ......
    |HNEXT| < HMIN
    Y:
        print: 'stepsize smaller than HMIN'
        (pause)
    N:
        H:=HNEXT
print:'ODEINT: more than NSTMAX points'
NVALUES:=NSTMAX
(return)
```

Structure chart 27 — RKQC(Y,F,N,X,HTRY,EPS,YSCAL,HNEXT)

```
SAFETY:=0.9
ERRCON:=0.0006
XSAV:=X
I=1(1)N
    YSAV(I):=Y(I)
    FSAV(I):=F(I)
H:=HTRY
STEPOK:=FALSE
    HH:=H/2.0
    RK4(YSAV,FSAV,N,XSAV,HH,YTEMP)
    X:=XSAV+HH
    DERIVS(X,YTEMP,F)
    RK4(YTEMP,F,N,X,HH,Y)
    X:=XSAV+H
    X = XSAV
    Y:
        print: 'RKQC: stepsize too small.'
        (pause)
    N:
        ......
    RK4(YSAV,FSAV,N,XSAV,YTEMP)
    ERRMAX:=0.0
    I=1(1)N
        ERRV(I):=Y(I)-YTEMP(I)
        TEMP:=| ERRV(I)/YSCAL(I)|
        ERRMAX < TEMP
        Y:
            ERRMAX:=TEMP
        N:
            ......
    ERRMAX > EPS
    Y:
        H:=SAFETY*H*
        EXP(-0.25*LOG(ERRMAX/EPS))
    N:
        STEPOK:=TRUE
        ERRMAX/EPS < ERRCON
        Y:
            HNEXT:=4*H
        N:
            HNEXT:=SAFETY*H*
            EXP(-0.2*LOG(ERRMAX/EPS))
        I=1(1)N
            Y(I):=Y(I)+ERRV(I)/15.0 (see comment P. 247)
STEPOK
(return)
```

Structure chart 28 — RK4(Y,G1,N,X,H,YOUT)

```
I=1(1)N
    YT(I):=Y(I)+H/2.0*G1(I)
XTEMP:=X+H/2.0
DERIVS(XTEMP,YT,G2)
I=1(1)N
    YT(I):=Y(I)+H/2.0*G2(I)
DERIVS(XTEMP,YT,G3)
I=1(1)N
    YT(I):=Y(I)+H*G3(I)
XTEMP:=X+H
DERIVS(XTEMP,YT,G4)
I=1(1)N
    YOUT(I):=Y(I)+H/6.0*(G1(I)+2*(G2(I)+G3(I))+G4(I))
```

6.5.4 Application of ODEINT+RKQC+RK4 to the ’satellite problem’.

The suite of programs described so far will now be applied to the ’satellite test’. The routine DERIVS corresponding to the differential system (6.22) has the form

```
void derivs(float x, float y[], float f[])
{
  f[1]=y[3];
  f[2]=y[4];
  f[3]=y[1]*y[4]*y[4] -alpha/y[1]/y[1];   // alpha GLOBAL = gamma*M
                                          //              = 1966.390
  f[4]=-2.0*y[3]*y[4]/y[1];
}
```

The results of this test are summarised in Fig.6.6. In this figure the exact elliptical orbit calculated analytically is compared with the Runge-Kutta results which are calculated with a program without (RUKUTEST) and with stepsize adaptation (ODEINT+RKQC+RK4).

Results:

- Program without stepsize adaptation (Fig.6.6, upper panel):

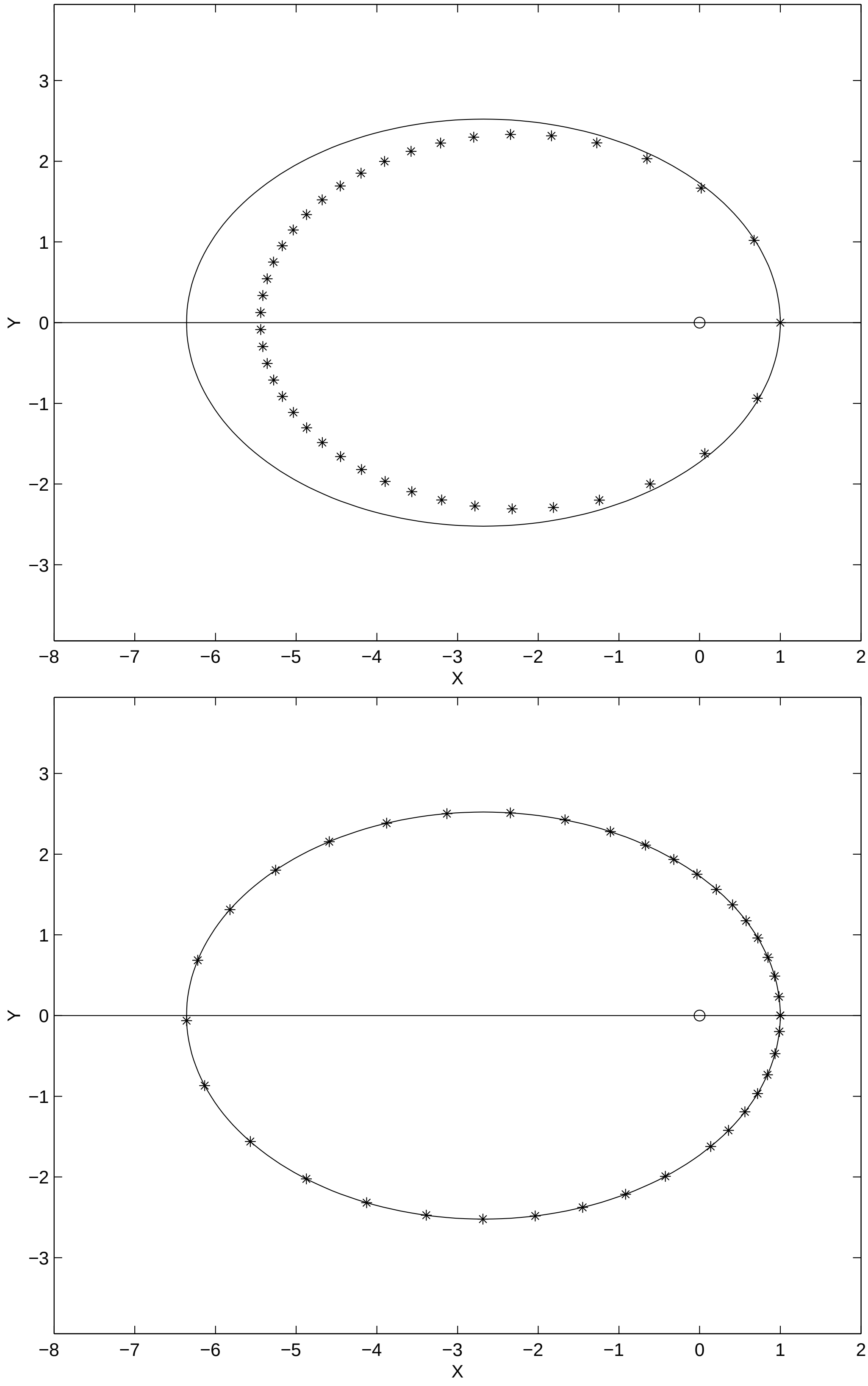


Figure 6.6: Efficiency of the stepsize adaptation in the 'satellite problem'. Comparison of the exact elliptical trajectory (full line) with the numerical values (stars). Above: Runge-Kutta *without* stepsize adaptation; Below: Runge-Kutta *with* stepsize adaptation.

In this test we assumed a constant stepsize $h = 1/50$ of the revolution period. The fact that h is constant *with time* has the following consequence: since a satellite moves more and more slowly the further it is from the earth (second Kepler's law), the points *in space* have a higher density, where the velocity is smaller. However, where the velocity is higher (and changes faster), i.e. when the satellite is closer to earth, the density of points is the smallest. But in this way the error is larger right where we would like it to be smaller!

The consequence of this bad point distribution is clearly seen in the diagram on the left: we see very large differences between the exact and the numerical trajectories.

- Program with interval amplitude optimisation, EPS=0.0001 (Fig.6.6, lower panel):

  In this case the density of points is maximal exactly where it is numerically sensible (i.e. in the region where the distance between the satellite and earth is the smallest). The effect is obvious: in case of a roughly equal number of points there is (at least graphically) no difference between the analytical and the numerical trajectory!

6.6 The Runge-Kutta-Fehlberg method.

The error estimate and adaptive stepsize algorithms of the Runge-Kutta suite of programs described in section (6.4.5) is very effective, but uses up a lot of computer power, because it requires two independent Runge-Kutta processes to run in parallel with each other.

Equally effective, but much less costly in terms of computer time is the following variant of the stepsize adaptive algorithm, which goes back to *Fehlberg*. We report here the formulas for the simple case of a differential system of order one [3]:

A move from x_0 to $x_0 + h$ through a Runge-Kutta method of order p gives:

$$\hat{y}^{(p)}(x_0 + h) = y_{exact} + Ch^{p+1}\,. \tag{6.29}$$

The same calculation, with a Runge-Kutta formula of next order $(p + 1)$, gives:

$$\hat{y}^{(p+1)}(x_0 + h) = y_{exakt} + \tilde{C}h^{p+2}\,. \tag{6.30}$$

The difference between (6.29) and (6.30) is:

$$\hat{y}^{(p)}(x_0 + h) - \hat{y}^{(p+1)}(x_0 + h) = Ch^{p+1} - \tilde{C}h^{p+2} \approx Ch^{p+1} \quad \text{for small } h$$

This can be solved for C:

$$C = \frac{\hat{y}^{(p)}(x_0 + h) - \hat{y}^{(p+1)}(x_0 + h)}{h^{p+1}}\,. \tag{6.31}$$

[3] Literature on the subject, see for example [10], P. 714ff, [23], P. 229ff

With this approximation for C we can estimate the error for a Runge-Kutta move of order p (6.29). In this way for each p we can determine an (ideal) interval amplitude h_{ideal}, corresponding to an error that <u>does not exceed</u> a given error threshold ϵ:

$$\epsilon = Ch_{ideal}^{p+1} = \left(\frac{h_{ideal}}{h}\right)^{p+1} |\hat{y}^{(p)} - \hat{y}^{(p+1)}|$$

or

$$h_{ideal} = h \left(\frac{|\hat{y}^{(p)}(x_0+h) - \hat{y}^{(p+1)}(x_0+h)|}{\epsilon}\right)^{-1/(p+1)} . \qquad (6.32)$$

Compare this result with Eq.(6.28).

To summarise the procedure in the *Runge-Kutta-Fehlberg method*: one calculates for the move $x_0 \to x_0 + h$ the approximate value $\hat{y}^{(p)}(x_0+h)$ and $\hat{y}^{(p+1)}(x_0+h)$. From this one estimates the 'optimal' stepsize through (6.32). There are now two possibilities:

1. if $h_{ideal} < h \quad \to \quad$ the move from x_0 to $x_0 + h$ is repeated with h_{ideal} ,
2. if $h_{ideal} \geq h \quad \to \quad$ the next move is performed with h_{ideal}.

What is the advantage in this method compared to the one described in section 6.4.5? As we know from the Runge-Kutta theory, there are in principle infinitely many valid Runge-Kutta formulas of a given order $p > 1$. <u>Fehlberg</u> was able to derive the following system of formulas:

$$\begin{aligned}
f_0 &= f(x_0, y_0) , \\
f_1 &= f(x_0 + \frac{h}{4}, y_0 + \frac{h}{4} f_0) , \\
f_2 &= f(x_0 + \frac{3h}{8}, y_0 + \frac{3h}{32} f_0 + \frac{9h}{32} f_1) , \\
f_3 &= f(x_0 + \frac{12h}{13}, y_0 + \frac{1932h}{2197} f_0 - \frac{7200h}{2197} f_1 + \frac{7296h}{2197} f_2) , \\
f_4 &= f(x_0 + h, y_0 + \frac{439h}{216} f_0 - 8hf_1 + \frac{3680h}{513} f_2 - \frac{845h}{4104} f_3) , \\
f_5 &= f(x_0 + \frac{h}{2}, y_0 - \frac{8h}{27} f_0 + 2hf_1 - \frac{3544h}{2565} f_2 + \frac{1859h}{4104} f_3 - \frac{11h}{40} f_4) .
\end{aligned}$$

With these definitions one can build a Runge-Kutta formula of fourth order, i.e.

$$\hat{y}^{(4)} = y_0 + h \left(\frac{25}{216} f_0 + \frac{1408}{2565} f_2 + \frac{2197}{4104} f_3 - \frac{1}{5} f_4\right)$$

but also a Runge-Kutta formula of fifth order, i.e.

$$\hat{y}^{(5)} = y_0 + h \left(\frac{16}{135} f_0 + \frac{6656}{12825} f_2 + \frac{28561}{56430} f_3 - \frac{9}{50} f_4 + \frac{2}{55} f_5\right) .$$

We thus obtain the following expression for the error:

$$\hat{y}^{(4)}(x_0+h) - \hat{y}^{(5)}(x_0+h) = -h \left(\frac{1}{360} f_0 - \frac{128}{4275} f_2 - \frac{2197}{75240} f_3 + \frac{1}{50} f_4 + \frac{2}{55} f_5\right) ,$$

and the calculation of the new ideal interval amplitude for p=4 through (6.32 gives

$$h_{ideal} = h \left(\frac{|\hat{y}^{(4)}(x_0 + h) - \hat{y}^{(5)}(x_0 + h)|}{\epsilon} \right)^{-1/5} . \qquad (6.33)$$

6.7 Other numerical methods for initial value problems.

To introduce the last part of this chapter, we cite a very colourful paragraph, taken from 'Numerical Recipes'[9]:

For many scientific users, fourth-order Runge-Kutta is not just the first word on ODE integrators, but the last word as well. In fact, you can get pretty far on this old workhorse, especially if you combine it with an adaptive stepsize algorithm ... Keep in mind, however, that the old workhorse's last trip may well be to take you to the stable: Bulirsch-Stoer or predictor-corrector methods can be much more efficient for problems requiring a very high accuracy. Those methods are the high-strung racehorses. Runge-Kutta is for ploughing the fields.

Indeed the methods that we have presented in detail (Runge-Kutta and Runge-Kutta-Fehlberg) are not the ultimate truth for the numerical solution of initial value problems. Nevertheless the author of this script decided it would be better to present a classical method (even though it is an old farm horse) in a very robust and usable suite of programs, rather than discuss the sometimes (not always) superior 'running horses' (Burlisch-Stoer methods and predictor-corrector methods). There is no space left in this context for a full treatment of these methods. Those who are interested should consult the following references:

- Approximate solution of selected differential equations: [19], P.127-228 (theory).
- Ordinary Differential Equations: [7], P.360-414 (as usual, strongly recomended: theory, numerical problems, comparison of methos, a.s.o.).
- Predictor-corrector methods (Adams-Bashford-Moulton): [2], P.220ff (Theory), P.428ff (Program).
- Extrapolation method of Bulirsch-Stoer: [2], P.233ff (Theory), S.434ff (Program).
- Richardson Extrapolation and the Bulirsch-Stoer Method: [9], P.563ff (short description and programs).
- Predictor-Corrector Methods: [9], P.569ff (short description).
- Predictor-Corrector method of Adams: [13], P.288f (Theory), P.290f (Algol-Program).
- Obviously all professional numerical-mathematical libraries contain many different programs.

6.7.1 Stiff sets of differential equations:

A few more remarks on the so-called *stiff system of differential equations*, which occur very often in practice. These are systems whose solutions comprise terms with very different dependencies on the independent variables. It is easier to illustrate them through an example (from [10], P. 734):

Let us assume that we want to solve the system

$$y_1'(x) = 998\, y_1 + 1998\, y_2$$

$$y_2'(x) = -999\, y_1 + 1999\, y_2$$

with the initial conditions $y_1(0) = 1$ und $y_2(0) = 0$.

The exact solution of this problem is

$$y_1(x) = 2\,\mathrm{e}^{-x} - \mathrm{e}^{-1000x} \qquad \text{and} \qquad y_2(x) = -\mathrm{e}^{-x} + \mathrm{e}^{-1000x}\,. \tag{6.34}$$

It is clear that the numerical calculation of such a problem with conventional methods (for example Runge-Kutta) leads to an *instability* of the algorithm, if one does not work with extremely small interval amplitudes [in practice, with $h < 1/1000!$]. This is particularly unfortunate, since the second terms in (6.34) decay to zero already for very small x, and from that point on, the solutions are therefore simply of the form $\exp(-x)$.

For this class of problems there are special methods of solution, such as the method of Kaps and Rentrop [4]. The corresponding programs with theoretical explanations can be found in [10], P. 734ff.

[4] P. Kaps und P. Rentrop, *Numerische Mathematik* **33**, P. 55ff.

Bibliography

[1] H. Ernst, *Numerische Methoden für Kleincomputer*, Luther-Verlag 1984.

[2] G. Engeln-Müllges und F. Reutter, *Formelsammlung zur Numerischen Mathematik mit Standard-FORTRAN-Programmen*, BI Mannheim 1984.

[3] G. Engeln-Müllges und F. Reutter, *Formelsammlung zur Numerischen Mathematik mit C-Programmen*, BI Mannheim 1990.

[4] G. Engeln-Müllges und F. Reutter, *Formelsammlung zur Numerischen Mathematik mit PASCAL-Programmen*, BI Mannheim 1985.

[5] G. Engeln-Müllges und F. Reutter, *Numerik-Algorithmen mit Programmen in FORTRAN, C und Turbo Pascal*, VDI-Verlag, 1996.

[6] G. Engeln-Müllges and F. Uhlig, *Numerical Algorithms with C or FORTRAN*, Springer-Verlag, 1996.

[7] W.S. Dorn and D.D. McCracken, *Numerical Methods with Fortran IV Case Studies*, Wiley 1972.

[8] A. Björck and G. Dahlquist, *Numerische Methoden*, Oldenburg Verlag 1972.

[9] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, *Numerical Recipes*, 2nd ed., Cambridge Uni Press 1992.

[10] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, *Numerical Recipes in C*, 2nd ed., Cambridge Uni Press 1992.

[11] M. Abramowitz and I.A. Segun, *Handbook of Mathematical Functions*, Dover Publications 1968.

[12] G.E. Forsythe und C.B. Moler, *Computer-Verfahren für lineare algebraische Systeme*, Oldenbourg Verlag 1971.

[13] G. Paulin und E. Griepentrog, *Numerische Verfahren der Programmiertechnik*, VEB Verlag Berlin 1975.

[14] P.K. MacKeown and D.J. Newman, *Computational Techniques in Physics*, Hilger 1987.

[15] I.S. Beresin und N.P. Shidkow, *Numerische Methoden 2*, VEB Verlag Berlin 1971.

[16] I.S. Beresin und N.P. Shidkow, *Numerische Methoden 1*, VEB Verlag Berlin 1970.

[17] Y.A. Shreider, *The Monte Carlo Method*, Pergamon Press Oxford 1967.

[18] R. Frühwirt, M. Regler, *Monte-Carlo-Methoden*, BI Mannheim 1983.

[19] B. P. Demidowitsch, L. A. Maron und E. S. Schuwalowa, *Numerische Methoden der Analysis*, VEB Verlag Berlin 1968.

[20] G. N. Poloshi, *Mathematisches Praktikum*, Teubner Verlag Leipzig 1963.

[21] Ch. Überhuber, *Computer-Numerik 1*, Springer-Verlag Berlin 1995.

[22] Ch. Überhuber, *Computer-Numerik 2*, Springer-Verlag Berlin 1995.

[23] P. L. DeVries, *Computerphysik*, Spektrum Akademischer Verlag Heidelberg, 1995.

Contents